Universitetet i Oslo
Institutt for lingvistiske fag

Hvem er du?
– Autentisering og formelle metoder

Olav Andreas Hoemsnes

Hovedfagsoppgave for graden Cand. Philol. i Språk, Logikk og Informasjon.

11. desember 2003

Forord

Jeg vil rette en stor takk til min veileder Herman Ruge Jervell som har vært en god inspirator gjennom arbeidet med denne oppgaven. Herman har funnet frem til interessant litteratur og har alltid tid og noen oppmuntrende ord å komme med. Miljøet ved Språk, logikk og informasjon har også hjulpet veldig i arbeidet med skriving og lesing av kryptiske tekster. Takk til alle medstudenter som har gjort hverdagen litt enklere med stadige ønsker om kaffepauser, lufteturer, middagsspising og turer på butikken. Jeg vil også takke mamma, pappa og lillesøster som alltid er der og som har meg i tankene når de vet jeg trenger det. Jeg må si unnskyld til alle dere jeg har forsømt den siste tiden, jeg skal prøve å skjerpe meg.

Til sist en stor takk til min kjære Andra som har stresset meg når det har vært nødvendig, og tatt vare på meg i de siste månedenes prøvelser.

Olav Andreas Hoemsnes.
Blindern desember 2003.

> If you think technology can solve your security problems, then you don't understand the problems and you don't understand the technology.
> (Bruce Schneier)

Innhold

Kapittel 1

Innledning

Dette arbeidet handler om autentiseringsprotokoller og ulike formelle metoder som benyttes i arbeidet med dem. I oppgaven benytter jeg de metodene ASM (Abstrakte tilstandsmaskiner), og AsmL (Abstrakt tilstandsmaskinspråk) for å lage en kjørbar modell av Needham-Schroeder protokollen for autentisering. Modellen er en spesifikasjon av protokollen og kan brukes som utgangspunkt for videre arbeid i ulike retninger. Den er nyttig i utviklingsarbeid, testing og vedlikehold av implementasjonen.

1.1 Oversikt

Arbeidet er strukturert i 11 kapitler og ett tilleg. I kapittel 2 kommer en drøfting om datasikkerhet basert på følgende hovedspørsmål. Hva legger vi i begrepet datasikkerhet? Hvorfor trenger vi datasikkerhet og hvordan blir vi påvirket av ulike krav om datasikkerhet? Jeg beskriver noen grunnleggende tjenester som regnes som basis for datasikkerhet. Tjenestene handler henholdsvis om *konfidensialitet*, *integritet* og *tilgjengelighet*. Deretter introduserer jeg et av kjernebegrepene i oppgaven, *autentisering*: Hva er autentisering for oss i dagliglivet og hva er det i dataverdenen? Deretter følger en kort drøfting om hvorfor den menneskelige faktoren er avgjørende i arbeidet med datasikkerhet.

Kapittel 3 gir en innføring i kryptografi. Med kryptografi tenker vi på teknikker for å gjøre skrift uleselig for andre. Vi ser hvordan utviklingen av kryptografiske teknikker har foregått, hvordan de brukes i dag og hvordan de danner grunnlaget for autentisering over datanettverk.

Kapittel 4 beskriver hva en protokoll er, og hvilke ulike typer av protokoller som finnes. Vi ser noen eksempler på ulike typer av protokoller og går igjennom hva som skjer ved bruk av en autentiseringsprotokoll.

I kapittel 5 går vi igjennom hvilke feil og angrep som kan rettes mot protokoller generelt, og autentiseringsprotokoller spesielt. Angrep og feil deles inn i ulike kategorier og det gir noen eksempler på de ulike feilene og angrepene som kan opptre. Vi ser også kort på hva

vi mener med en angriper i et datanettverk. Hvilke handlinger kan angriperen foreta, og hvilke ressurser er angriperen i besittelse av?

Kapittel 6 viser de ulike formelle metodene som brukes i arbeidet med autentiseringsprotokoller. Hvorfor bruker vi formelle metoder for å designe og studere autentiseringsprotokoller? Vi deler de formelle metodene inn i ulike kategorier og ser nærmere på hver enkelt. Fordeler og ulemper ved de ulike metodene blir diskutert og vi kommer kort inn på hva som kan være fornuftig anvendelse.

Needham-Schroeder protokollen for autentisering beskrives i kapittel 7. Vi går gjennom noen av de feil og angrep som er rettet mot den og hvordan den er blitt forbedret. Vi kommer tilbake til protokollen i min AsmL spesifikasjon av den.

Kapittel 8 beskriver nærmere Abstrakte tilstandsmaskiner (ASM), som er en av de formelle metodene som kan brukes i arbeidet med autentiseringsprotokoller. Vi sier litt om hvorfor en ønsker å bruke ASMer, og gir eksempler på hvordan en kan gå frem for å lage en ASM. Bruken av ASM i akademia og i næringslivet beskrives kort. Verktøy som kan benyttes i arbeidet med ASM blir også nevnt, i det neste kapittelet tar vi nærmere for oss et av disse verktøyene.

Verktøyet i kapittel 9 AsmL, er et kjørbart spesifikasjonsspråk som bygger på ASM. I kapittelet beskrives AsmL systemet og hvordan det kan brukes i design og utvikling av protokoller, programmer og maskinvare. Vi ser på noen enkle eksempler på bruk av AsmL, før vi går løs på en spesifikasjon av Needham-Schroeder protokollen for autentisering i AsmL.

I kapittel 10 beskrives en modell for en enkel versjon av Needham-Schroeder protokollen for autentisering som jeg har laget. Vi ser på de ulike delene den består av og hvordan den kan kjøres. Arbeidet med spesifikasjonen kan sees på som en utviklingsprosess, hvor vi starter med en enkel beskrivelse og går trinnvis over til en detaljert spesifikasjon. Videre ser vi på hva som kan læres av arbeidet med en slik spesifikasjon, og hvilke muligheter vi har for videre arbeid. Det videre arbeidet kan gå i flere retninger og vi ser nærmere på disse retningene. Vi oppsumerer arbeidet som er gjort og ser på resultatene av det.

1.1.1 Begrepsavklaring

I oppgaven bruker jeg ordene modell og spesifikasjon litt om hverandre. Dette fordi jeg forstår en modell som noe, som også kan være en spesifikasjon. Dette er en ordbruk som også går igjen innen fagfeltet. En modell kan være en spesifikasjon, og en spesifikasjon kan være realisert i form av en modell. Å si når en modell blir en spesifikasjon, kan noen ganger være vanskelig å bestemme.

Kapittel 2

Sikkerhet i et bredere perspektiv

2.1 Hvorfor sikkerhet?

I dagens samfunn blir vi stadig møtt av overskrifter om hvordan teknologien knytter oss sammen på den ene siden, og hvordan den fremmedgjør på den andre. Vi har eksempler på bruk av datamaskiner og datanettverk for å kunne utføre mange ulike oppgaver som vi tidligere gjorde ved hjelp av samtaler og møter ansikt til ansikt. Dette stiller nye krav til hvordan vi kommuniserer og ikke minst hvordan vi etablerer tillit til banker, institusjoner og andre mennesker. Vi trenger verktøy og hjelpemidler for å kunne fastslå identiteten til andre mennesker, til andre firmaer, og institusjoner på en sikker, god og enkel måte. De fleste kjenner vel til tanker som "Hva var nå dette passordet?" eller "Hvilken pin-kode skal jeg bruke nå". Alle disse pin-kodene og passordene som vi bruker sammen med et brukernavn eller et bankkort er forsøk på å bevise sin egen identitet, eller besittelse av noe f.eks.. penger overfor en annen part. Denne andre parten kan være et menneske, program eller firma. Det å bevise sin identitet overfor andre kaller vi *autentisering*.

Det første vi ofte gjør når vi kommer på jobb eller skole er å logge på en datamaskin. Vi bruker da normalt et brukernavn i kombinasjon med et passord. Det er et eksempel på bruk av noe vi vet. Et brukernavn (som ofte er offentlig kjent) og et hemmelig passord. Andre former for identifisering baseres ofte på noe vi er i besittelse av; det kan være et smartkort, eller en annen form for "ting" (token). Andre mekanismer benytter seg av noe vi er, så kalt biometrics. Det kan være fingeravtrykk, håndavtrykk, iris-skanning o.l. De aller sikreste systemene benytter seg av kombinasjoner av alle disse tre. Det vil si noe vi vet, noe vi har og noe vi er.

For å kunne gjøre autentiseringsoperasjoner overfor mange ulike aktører er vi avhengig av å gjøre det på samme standardiserte måte hver gang og til det bruker vi protkoller. En protokoll er en slik fast "oppskrift" på hvordan vi gjør noe. Vi kommer tilbake til protokoller for autentisering i kapittel 4. Alle de ulike måtene vi prøver å beskytte vår identitet på er utsatt for angrep, de er ofre for feil i metodevalg, eller de kan være for omstendelige. Vi har ulike nivåer av sikkerhet for ulike typer av transaksjoner og har helt andre krav for store

transaksjoner enn små. Det er helt ulike toleransenivåer for en handel på 10 000,- kroner, enn det er for noe som koster 10,- kroner. I slike sammenhenger blir det kjente uttrykket "tid er penger" så absolutt gjeldende.

Eksempler på sikkerhetsbrudd blir publisert og oppdaget hver dag. Vi møter stadige oppslag om nye virus, ormer m.m som truer datamaskiner og nettverk. De fleste slike meldinger kommer som en følge av at feil og mangler i implementasjonen av protokoller og programmer blir utnyttet. Disse feilene kan være alt fra rene logiske feil til de mer tekniske feilene. Vi kommer tilbake til feil og angrep i forbindelse med protokoller i kapittel 5.

2.2 Elementene i et sikkert system

Som brukere av datamaskiner og nettverk er vi avhengige av et system som kan tilby *tilgjengelighet*, *integritet* og *konfidensialitet*. I følge Bishop er hver av disse tjenestene hovedkomponenter i et sikkert system. La oss se litt nærmere på disse tre:

2.2.1 Konfidensialitet

Konfidensialitet er det å holde informasjon eller ressurser hemmelig. Grunnen til dette er at datamaskiner ofte brukes til å lagre sensitiv informasjon for militæret, det offentlige og for privatpersoner. Alle sykehus må f.eks.. holde pasientjournaler hemmelig. De vanligste måtene å sikre konfidensialitet på er igjennom tilgangskontroll og kryptering av materiale. Det fungerer slik at informasjonen enten er skjult for brukere som ikke skal ha tilgang til den, og/eller at informasjonen er gjort uleselig for brukere som ikke skal kunne lese den. Problemet med den første måten å gjøre det på er at brukere kan lese informasjonen om de kommer seg rundt tilgangskontrollen på en eller annen måte. Noen ganger kan det til og med være nødvendig å sikre konfidensialitet på den måte at informasjon eller ressursen ikke blir gjort synlig. Eksistensen av informasjonen eller ressursen skal beskyttes. Det er noe som f.eks.. gjelder for datanettverk/systemer som skjuler systeminformasjon for å gjøre eventuelle angrep vanskeligere å gjennomføre.

2.2.2 Integritet

Integritet henviser til hvorvidt data og ressurser er til å stole på eller ikke. Integritet finnes på to nivåer; *dataintegritet* (selve innholdet) og *opphavsintegritet* (kilden til dataene, autentisiteten).Vi vil sikre oss at data ikke blir forandret av utenforstående og vi vil ha kontroll på datakilden. Vi vil vite om dataene kommer fra en sikker kilde eller ikke. De vanligste mekanismene for å sikre dette er: forebygging og forhindringsmekanismer og oppdagingsmekanismer.

Forebygging og forhindring prøver å beskytte data og ressurser gjennom å blokkere uautoriserte forsøk på å forandre data og å forhindre autoriserte forsøk på uautoriserte endringer. Det første er når brukere som ikke skal ha tilgang til data prøver å gjøre forandringer på de.

Det kan hindres igjennom tilgangskontroll. Det andre er når en person som har tilgang til og mulighet til å gjøre forandringer på dataene gjør forandringer som han/hun ikke har lov til å foreta. Det siste er vanskeligere å forhindre enn det første. Et eksempel på dette var en australsk ansatt i et stort flyselskap som tok ut bonuspoeng fra reisende som ikke hadde registrert de og overførte de til seg selv. Han kunne deretter bruke disse til å reise som han ville. Dette er noe som er både enkelt og vanskelig på samme tid. Det kan være vanskelig å gardere seg mot siden det er en på innsiden som utfører "angrepet". Autentisering og aksesskontroll kan forhindre de fleste forsøk på uautoriserte endringer utenfra. Dersom det er noen på innsiden er det vanskeligere å forhindre og krever andre mekanismer.

Oppdagelsesmekanismene fungerer mer som bokføring og logging av hva som blir gjort med data og ressurser. De kan brukes til å se i etterkant om data er forsøkt forandret eller ikke. Slike mekanismer fungerer på ulike måter, noen logger bruker og systemaktivitet, andre bruker føringer som mål på hvordan data skal se ut. På grunn av alle poengene som den ansatte i eksempelet ovenfor fikk lurt til seg, hadde han så mange bonuspoeng at han for å kvalifisere til disse måtte ha reist på 2-3 lange flyturer hver uke i flere år. Dette burde kanskje fått noen alarmklokker til å ringe. En melding kunne eventuelt blitt sendt til en som kontrollerte poenguttak dersom de oversteg en viss sum. Det ville være et eksempel på en oppdagelsesmekanisme.

Integritet omfatter både hvorvidt data er til å stole på, er kilden til data sikker? Og det omfatter om selve dataene er korrekt, er det brukt riktig nøyaktighet i tall osv. Integriteten til data påvirkes av flere faktorer slik som hvor de kom fra, hvordan de oppbevares osv. og på grunn av dette kan integriteten til data være vanskelig å bestemme.

2.2.3 Tilgjengelighet

Tilgjengelighet skal sikre oss tilgang til informasjon og programmer når vi måtte ha bruk for det. Dette er en forutsetning for arbeid og drift. Det er ingen hjelp i å ha en fin nettbutikk dersom den er utilgjengelig for kunder pga. tekniske problemer, angrep, virus eller lignende. Sikkerhet i forhold til tilgjengelighet kommer inn i bildet når vi vet at angrep kan rettes mot selve tilgjengeligheten til en tjeneste. Slike angrep kalles ofte DoS-angrep[1]. De kan være vanskelig å avdekke, siden de ofte fungerer på sammen måte som økt trafikk på et nettsted. Dette er noe som forekommer f.eks.. i forbindelse med kampanjer på flybilletter hvor flyselskap opplever at servere og bookingsystemer ikke klarer å ta unna den økte trafikken og derfor går ned. Dette kan noen ganger forveksles med et DoS-angrep, og vice versa.

Konfidensialitet, integritet og tilgjengelighet handler på mange måter om en ting; kontroll av tilgang til informasjon og data. Disse tre essensielle tjenestene er grunnlaget for datasikkerhet slik vi tenker på det i dag. I denne oppgaven skal vi se nærmere på

[1] DoS er en engelsk forkortelse for Denial of service

autentisering. Autentisering er en forutsetning for å styre tilgangen, så kalt aksess-kontroll, til data og informasjon.

2.2.4 Autentisering

Fra bokmålsordboka:

```
autentisk - auten´tisk a2 (gr authentikos, av authentes 'opphavsmann')
fullt ut ekte og pålitelig en a- beretning / a- folkemusikk
/ a-e brev / et a- intervju .
```

Vi utfører identifisering og autorisasjon av folk og tjenester utallige ganger hver dag. Det er noe som skjer så ofte at det blir nærmest automatisk. Som mennesker baserer vi oss på flere sanser for å kunne stole på om andre mennesker er de vi tror de er. Politifolk har spesielle uniformer som vi kjenner igjen og gjør at vi behandler de på spesielle måter. Dersom vi kjøper pølser i en pølsekiosk, ser vi på, lukter og smaker at dette er faktisk en ekte pølse. Vi kan avsløre det umiddelbart om pølseselgeren forsøker å selge oss "banan i brød" istedenfor "pølse i brød". Dersom du går på fotballkamp er det en egen stemning og egne lyder som bekrefter overfor deg at dette er en ekte fotballkamp. Det ville være veldig ressurskrevende å arrangere en "fiktiv" fotballkamp for å lure noen. Ressursene som brukes kommer an på målet med lureriet jf. filmen "The Sting" [2]. Lureriet i filmen fungerer, men det krever store ressurser og mye arbeid for å få det til. I dataverdenen er slike lureri mye enklere å gjennomføre. Det skal ikke så mye jobb til for å utgi seg for å være en nettbutikk. Det finnes flere eksempler på falske nettbutikker, og ekte nettbutikker som er overtatt av utenforstående, for å kunne få tak i kredittkortnumre og personlige opplysninger for å svindle til seg penger. Selv om mediet er et annet, foregår slik svindel på omtrent samme måte som utenfor "cyberspace", forskjellen ligger i hvor enkelt det er å gjøre det, og hvor store ressurser som må til.

Det er veldig enkelt å få til identifikasjon og autentisering ansikt-til-ansikt fordi vi har så mange trekk å gå etter. Vi ser personens ansikt, kroppsspråk, hører stemmen og kan kjenne igjen lukten. Dette fungerer ikke på samme måte over internett. For eksempel når vi skal autentisere oss overfor en datamaskin, eller et datanettverk. Noe som kanskje er enda vanskeligere er hvordan vi skal få et program eller system til å kunne overbevise oss om at det er virkelig det programmet eller systemet vi ønsker å benytte.

2.2.5 Systemer

Tjenestene som vi har beskrevet ovenfor er noe vi ønsker at systemene vi bruker skal kunne gi oss. Hva er det som gjør at dette er så vanskelig? Det er særlig fire egenskaper til systemer som skaper problemer:

[2] "The sting" eller "Stikket" som den heter på norsk, er en amerikansk film fra 1973. I filmen svindler to gangstere en annen storgangster for å ta hevn for mordet på en venn. De setter opp en "falsk" bookmakersjappe med fiktive klienter, hesteveddeløpsoverføringer m.m.

- Kompleksitet. Systemer som vi bruker blir stadig mer komplekse. Selv enkle programmer inneholder etterhvert flere hundre tusen linjer med kode. Systemer inneholder kanskje hundrevis av komponenter som skal fungere problemfritt sammen og for seg selv.
- Interaksjon. Flere og flere systemer skal fungere sammen med andre systemer. I dag er de aller fleste systemer, store som små, på en eller annen måte knyttet til internett. Noen ganger vet vi om det, andre ganger ikke.
- Uforutsigbarhet. Etterhvert har det vist seg at komplekse systemer har egenskaper som ikke var forutsett. De oppstår. Dersom vi har ett sikkert system og kobler det sammen med et annet sikkert system, er det ikke nødvendigvis slik at det resulterende systemet er sikkert. 2 + 2 er ikke alltid 4 i denne sammenhengen.
- "Bugs". Systemer inneholder feil. Slike "bugs" omfatter alle typer feil, fra rene programmeringsfeil til uforklarlig oppførsel (se forrige punkt).

2.2.6 Teknologi og den menneskelige faktoren

I denne oppgaven kommer vi til å se nærmere på en del tekniske sider ved kryptering, autentiseringsprotokoller, formelle metoder og verktøy som brukes i forbindelse med disse. En veldig viktig ting som ikke blir tatt opp er den menneskelige faktoren. I god tradisjon med informatikere og teoretikere verden over vil vi behandle brukerne som "maskiner", vi forutsetter at de ikke gjør feil. Det er en selvfølge at feil bruk av en autentiseringsprotokoll, kryptografisk funksjon eller hva det måtte være, fører til sikkerhetsproblemer. Selv om vi beviser at funksjonene vi bruker i kryptering er umulige å knekke, protokollen er uangripelig og implementasjonen er riktig, kommer vi ikke unna det menneskelige elementet. Det er tross alt mennesker som bruker disse funksjonene på en eller annen måte. Den enkleste måten å tilegne seg et brukernavn og passord fra noen, er å ringe og spørre om det. En av de mest kjente "hackere" i USA, Kevin Mitnick, var veldig god til å utnytte menneskers gode tro. På engelsk kalles det "social engineering"; det vil si å utnytte det faktum at vi som mennesker ofte stoler på andre menneskers ord. Den vanligste måten å utføre slike "angrep" på er å ta en telefon til noe som jobber i organisasjonen vi ønsker å bryte oss inn i. Vi utgir oss for å være en ansatt i IT-avdeling, eller en annen autoritetsperson og spør etter brukernavn og passord til denne personen. Veldig mange mennesker faller for slike telefonsamtaler. I en stor organisasjon kjenner ikke alle ansatte hverandre, noe som gjør jobben enklere. Vi kommer ikke utenom menneskene som bruker teknologien vi ønsker å sikre. Vi kommer ikke til å fokusere på den menneskelige faktoren i denne oppgaven, men den må absolutt nevnes.

Selv om datasikkerhetsselskaper og informatikere ofte fokuserer på teknologi, må vi ikke glemme at vi ofte opererer i en gråsone, kontekst er ofte viktigere enn teknologi. Med kontekst tenker vi på alle tingene som kommer i tillegg til selve "sikkerhetsteknologien". Her er den menneskelige faktoren kanskje den viktigste.

Vi har sett på de utfordringer vi møter i å flytte stadig flere funksjoner over på datasystemer og nettverk. Hvordan kan vi ta vare på den sikkerheten vi føler vi har krav på i dagliglivet. Hvordan skal vi opprettholde egen identitet og kunne handle på internett like sikkert som hos butikken på hjørnet. Hvilke utfordringer er knyttet til bruken av stadig mer komplekse systemer, og hva legger vi i begrepet datasikkerhet? Når det gjelder de ulike egenskapene vi knytter til datasikkerhet har vi særlig tatt for oss autentisering. Et hjelpemiddel for å sikre autentisering over datanettverk er bruken av kryptografiske teknikker. Neste kapittel gir en kort innføring i kryptografi.

Kapittel 3

Kryptografi

3.1 Introduksjon

Ordet kryptografi er avledet fra gresk og betyr "skjult skrift". For å forstå datasikkerhet er det nødvendig å forstå kryptografi. Det er ikke nødvendig å forstå all matematikken, men prinsippene bak de teknikkene som brukes og hvilke følger de får. Vi trenger å vite hva kryptografien gjør oss i stand til å gjøre og hva den ikke gjør oss i stand til. Ofte bruker en ordet *kryptere* om å gjøre tekst uleselig, og *dekryptere* om å gjøre den krypterte teksten om til vanlig tekst igjen. Dersom noen uvedkommende prøver å gjøre teksten leselig uten tilgang på nøkler og/eller algoritme dreier det seg om *kryptoanalyse*.

Tradisjonelt har kryptering blitt brukt for å gjøre informasjon uleselig for andre. Ved å ta utgangspunkt i en s.k. klartekst, som er informasjonen i sin originale form, har en ved hjelp av en krypteringsalgoritme gjort teksten uleselig. Når informasjonen skal leses igjen blir den dekryptert, ofte ved hjelp av en bestemt nøkkel. Vi skal se nærmere på hvordan dette er blitt gjort før i tiden, og hvilke kryptografiske teknikker som benyttes i dag. Kapitlet inneholder en del bakgrunnsinformasjon som er nødvendig for å forstå de kryptografiskeprotokollene og spesielt autentiseringsprotokollene som kommer i de neste kapitlene (kapittel 4.7og 7).

3.2 Klassisk kryptografi

Det har alltid vært behov for å holde informasjon hemmelig. Overgangen fra en muntlig til en skriftlig kultur gjorde at informasjon kunne bevares "for evig" og det var ikke sikkert at alle ville at informasjon som ble nedskrevet skulle kunne leses av andre til evig tid. Særlig ble hemmeligholdelse viktig i et militært perspektiv hvor meldinger om angrep, troppeforflyttninger o.l. skulle holdes hemmelig. Det ble derfor nødvendig å kunne gjøre skriften uleselig for eventuelle motstandere som fikk tak i meldingen. Noen av de første eksemplene har vi fra romerriket hvor Herodotus, ifølge historikeren Cicero, beskriver hvordan en hemmelig melding ble sendt under krigen mellom Hellas og Persia (Singh 1999). Vi har også

eksempler på kryptering fra det gamle norrøne runealfabetet f.eks.. på Rökstenen utenfor Rök kirke i Östergötland (Johnsen 2001).

3.2.1 Substitusjon

3.2.1.1 Caesar-chifferet

En av de første krypteringsalgoritmene har fått navn etter keiseren som benyttet den, nemlig Caesar-chifferet. Selve Caesar-chifferet illustrerer en av de fundamentale operasjonene som all senere kryptering bygger på (Singh 1999). Krypteringsalgoritmen er veldig enkel. Den består i å ta utgangpunkt i alfabetet og "forskyve det" et bestemt antall plasser. Slik at vi får rekken

```
abcdefghijklmnopqrstuvwxyzæøå
ghijklmnopqrstuvwxyzæøåabcdef
```

noe som vi kan bruke som algoritme for kryptering. Dersom vi skal kryptere ordet "mandag" blir det utifra Caesar-chifferet vi har satt opp "sgtjgm". Dette er et eksempel på en av de enkleste teknikkene innen kryptering, s.k. *substitusjon*. Når vi bruker kun et alfabet vil algoritmen være en såkalt monoalfabetisk substitusjon. Selv om denne algoritmen virker veldig simpel, var det på denne tiden ikke mange som kunne lese og skrive, og dette gjorde algoritmen relativt sikker mot utenforstående.

Til tross for at denne formen for kryptering var relativt sikker i sin spede barndom, var den rask å knekke når kodeknekkerne fikk øynene opp for frekvensanalyse. I alle språk er det noen bokstaver som går igjen oftere enn andre. Dersom en lager lister over hvor ofte bestemte bokstaver forekommer i vanlige tekster, er det fort gjort å knekke en kode når den består av en tekst på en viss størrelse. Desto større tekst å utføre frekvensanalysen på, jo enklere er det å finne de riktige substitusjonene av bokstaver eller tegn.

3.2.1.2 Vigenère-chifferet

For igjen å komme i forkant av kodeknekkerene var det etterhvert nødvendig å finne enda bedre algoritmer for kryptering. Den neste store nyvinningen i så måte var Vigenère-chifferet. Vigenère-chifferet har fått sitt navn etter den franske vitenskapsmannen Blaise de Vigenère som jobbet for det franske hoffet til Henry den tredje (16. århundre). Vigenère-chifferet var konstruert slik at hver bokstav ikke ble kodet som samme bokstav eller tegn også neste gang det forekom i teksten. Dette gjør det mye vanskeligere å utføre en frekvensanalyse av den krypterte teksten. I stedet for å bruke kun et alfabet slik som i Caesar-chifferet ovenfor, brukes flere alfabeter. Denne formen for kryptering er derfor et eksempel på en polyalfabetisk substitusjon. En substitusjon som er utført med flere alfabeter. Det kan sammenlignes med å bruke en nøkkel, noe vi skal se nærmere på etter figur 3.2.1. Vigenère kan settes opp som en tabell (s.k. tablå):

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Figur 3.2.1: Vigenère tablå

For å kryptere en setning i Vigenère-chifferet må en bruke et såkalt *nøkkelord*. Dette nøkkelordet er det samme som en nøkkel. Dersom vi skal kryptere setningen: "Angrepet starter mandag" og nøkkelordet er "tirsdag" blir krypteringen som følger:

```
nøkkelord          : tirsdagtirsdagtirsdag
klartekst setning  : angrepetstartermandag
kryptert setning   : tvxjhpkmalsutjkurfgam
```

Her er ordene i setningen satt sammen uten noen form for tegnsetting, for å gjøre kryptoanalyse vanskeligere. For å dekryptere meldingen er det bare å plassere nøkkelordet over den krypterte teksten og finne den opprinnelige meldingen ved å "slå opp" i tablået (se figur 3.2.1). Vigenères kryptosystem representerer et av kryptologiens historiske høydepunkter (Johnsen 2001). Vigenères tablå er som addisjonstabellen:

$$\text{chiffer} = (\text{klartekst} + \text{nøkkel}) \bmod 20$$

Denne krypteringen ble sett på som absolutt sikker i flere hundre år. Faktisk ble Vigenère-chifferet benyttet helt fram til 1863 uten at noen hadde funnet en systematisk metode for å knekke den. Da fant den preusisske majoren Kasiski en metode for å knekke Vigenère-chifferet. Metoden hans går ut på å analysere ord med to bokstaver, tre bokstaver, osv. (bigram, trigram osv. på engelsk). Dersom vi kan finne slike i den krypterte teksten, er det mulig å bruke avstanden mellom disse til å finne nøkkelordet. Og dersom vi finner nøkkelordet er det bare å anvende det til å dekryptere den krypterte teksten.

En ting som er veldig interessant ved Vigenère-chifferet er bruken av en nøkkel. Nå var det ikke lenger nødvendig å holde krypteringsalgoritmen hemmelig, så lenge nøkkelen var hemmelig. Dette er en viktig forutsetning i moderne kryptografi. En krypteringsalgoritmes styrke ligger ikke i hemmeligholdelse av algoritmen, men i hemmeligholdelse av nøkkelen. Det gjør at vi kan ha allment kjente krypteringsalgoritmer som kan studeres av eksperter og forbedres dersom de inneholder feil. Det kan gjøres uten noen trussel mot hemmeligholdelsen av en melding som er kryptert med algoritmen. Hemmeligheten er beskyttet av nøkkelen og ikke algoritmen.

3.2.2 Transposisjon

Vi har sett på en av de mest brukte teknikkene innenfor kryptering; substitusjon. En annen mye brukt teknikk er *transposisjon*. En algoritme som benytter seg av transposisjon bytter plass på bokstavene, selve tegnene eller bokstavene blir ikke endret (substituert). Vi endrer rekkefølgen på tegnene etter bestemte mønstre. På godt norsk vil vi kalle det en omflytting eller omstokking av rekkefølgen på tegnene. Slike systemer kalles også omkastningssystemer. Noen enkle eksempler på algoritmer som benytter seg av denne teknikken kan være:

* Revers chifferet
* "Rail fence" chifferet
* Skytale chifferet

3.2.2.1 Revers chifferet

Det første chifferet er veldig enkelt å vise. Vi tar tekststrengen og snur om på rekkefølgen.

```
‘‘vi møtes til lunsj’’ blir:
‘‘jsnul lit setøm iv’’
```

3.2.2.2 "Rail fence" chifferet

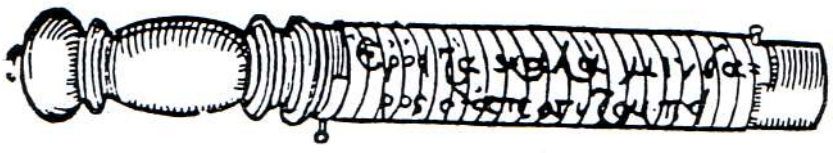
"Rail fence" er en litt annen omkastning. Teksten: "vi møtes til lunsj" settes opp slik:

```
v   m    t    s   i   l   n   j
  i   ø    e    t   l   u   s
```

og vi får chifferteksten: "vmtsilnjiøetlus" som dersom den stilles opp riktig dekrypteres til "vimøtestillunsj".

3.2.2.3 Skytale chifferet

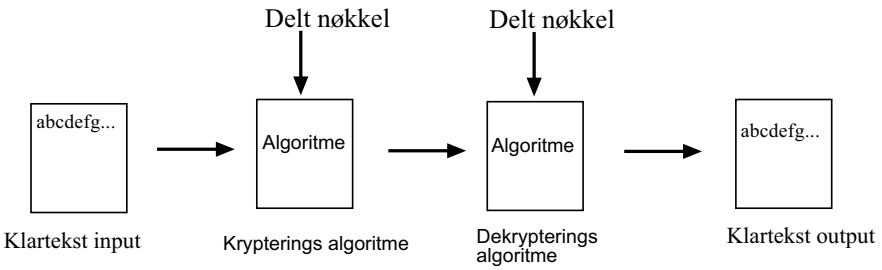
Skytale er navnet på et chiffer som ble brukt av regjeringen i Sparta for nesten 2500 år siden. Den baserte seg på en omkastning som ble gjort på følgende måte; en lang skinnreim ble viklet rundt en stokk (Skytale) se figur 3.2.2, og teksten som skulle krypteres ble skrevet på langs av stokken, dvs. på tvers av skinnreima. Resten av skinnreima ble fylt med andre bokstaver uten mening. Når skinnreima ble viklet av igjen var teksten på skinnreima uleselig, hvis ikke den ble viklet rundt en stokk av nøyaktig samme omkrets. Disse sylinderne kunne også ha ulik omkrets for å brukes til ulike krypteringsoppgaver (Johnsen 2001).



Figur 3.2.2: Skytale Illustrasjon fra Porta 1539

3.3 Symmetrisk kryptografi

Mye av det vi har sett på nå er eksempler på det vi kaller *symmetrisk kryptografi*. Symmetrisk kryptografi er bruken av en og samme hemmelige nøkkel, for å kryptere og dekryptere en og samme melding. Dersom to personer, Alice og Bob, ønsker å kommunisere, kan de bli enige om en hemmelig nøkkel. Denne nøkkelen bruker så Alice for å kryptere en melding, og sender denne til Bob. Bob bruker den samme nøkkelen for å dekryptere og lese meldingen. Dette kan vi se i figuren under, hvor en og samme nøkkel blir brukt for å kryptere og dekryptere en melding vha. DES algoritmen.



Figur 3.3.1: Symmetrisk kryptering

Dette kan gjøres med velkjente, og så langt vi vet, sikre kryptoalgoritmer. Som tidligere nevnt ligger sikkerheten til kryptosystemet i hemmeligholdelse av nøkkel og ikke algoritme. En måte å knekke slike koder er å prøve alle mulige nøkler. Derfor er valget av nøkler veldig viktig. En nøkkel bør være av en viss lengde, i dag brukes det typisk nøkler som er på 128 bits i mange nettlesere.

Eksempler på moderne kryptoalgoritmer er tyskernes Enigma brukt under 2. verdenskrig og DES, 3DES og AES.

Disse kryptoalgoritmene benytter seg av teknikker som vi allerede har sett på; substitusjon og transposisjon. Dette gjøres flere ganger etter hverandre, i tillegg til å benytte seg av andre mer spesielle teknikker. Hvis det er slik at vi kan bruke lange nøkler og gode krypteringsalgoritmer, hva er så problemet her? Et av de største problemene ligger i utvekslingen av nøkler. Så lenge Alice og Bob må møtes for å bli enige om den felles hemmelige nøkkelen har vi ikke kommet så veldig langt. Vi er avhengige av en løsning på problemet med *nøkkeldistribusjon*.

3.3.1 DES

Vi skal se litt nærmere på en mye brukt algoritme i nyere tid nemlig DES. DES står for Data Encryption Standard, og det var først en amerikansk senere internasjonal standard. Standarden ble initiert av NBS [1] i 1972 og utviklet av IBM, NSA [2] m.fl. i en serie av

[1] National Bureau of Standards, nå NIST (National Institute of Standards and technology)

[2] NSA - National Security Agency, USA. Regnet som verdens fremste kryptorganisasjon, med verdens største samling av kryptoforskere og matematikere.

workshops. Den ble godkjent for bruk i 1976 og det kom som en overraskelse på NSA at den ble publisert på en slik måte at det var mulig å lage implementasjoner av den i programvare. NSA trodde den kun var ment til bruk i maskinvareimplementasjoner (Schneier 1996). For resten av verden og særlig i kryptomiljøer var det at en NSA "godkjent" algoritme var blitt tilgjengelig et stort skritt fremover. DES ble også godkjent som en privat standard i 1981. DES er en såkalt blokkchiffer, dvs. at klarteksten blir behandlet i blokker av tekst 64-bit av gangen. DES algoritmen bygger på en såkalt *Feistel*–struktur.

3.3.1.1 Feistel–strukturen

Feistel–strukturen (chifferet) er et system for hvordan klarteksten blir behandlet sammen med nøkkelen for å kryptere teksten. Input til algoritmen er tekstblokker på *2w* bits og en nøkkel K som blir delt i to deler R(høyre) og L(Venstre). Disse to delene av klarteksten går igjennom n runder av prosessering som kombineres for å danne den krypterte outputen. Hver runde har som input to deler, L_{i-1} og R_{i-1}, som er output fra foregående runde og en subnøkkel, K_i, utledet av nøkkelen K. Alle rundene har den samme strukturen. Først utføres en substitusjon på den venstre delen av dataene. Dette gjøres ved å bruke en *rundefunksjon* F på høyre del av data for så å ta exclusive-OR (XOR) på output av denne funksjonen og venstre halvdel av data. Rundefunksjonen har samme generelle struktur for hver runde modifisert av subnøkkelen for runden K_i. Etter denne substitusjonen, gjennomføres en permutasjon som utveksler de to halvdelene med data. Hvordan selve Feistel–strukturen blir realisert bestemmes blant annet av:

- Blokk størrelse: Større blokker gir økt sikkerhet (dersom alle andre parametre er like) men reduserer hastigheten på kryptering/dekryptering.
- Nøkkel størrelse: Større nøkler gir økt sikkerhet, men kan også her redusere hastigheten på kryptering/dekryptering. Den mest brukte nøkkellengden i moderne algoritmer er 128 bit. Vi skal se nærmere på nøkkellengde litt senere i kapitlet.
- Antall runder: Essensen i Feistel chifferet er at en runde gir dårlig sikkerhet, men flere runder gir økt sikkerhet. Typisk antall runder er 16.
- Algoritme for generering av subnøkler: Større kompleksitet i denne algoritmen, gir økt vanskelighet for kryptoanalyse.
- Runde funksjon: Som for subnøkkel generering gir økt kompleksitet større motstand mot kryptoanalyse.

Dekryptering vha. Feistel–chifferet er veldig likt det som skjer ved kryptering. Vi bruker chifferteksten som input til algoritmen, men bruker subnøklene i motsatt rekkefølge, dvs. vi starter med K_{16} i første runde og avslutter med K_1 i siste runde. Det elegante med denne måten å gjøre det på, er at vi kan benytte samme algoritme både for kryptering og dekryptering.

3.3.1.2 DES–algoritmen

Klarteksten er på 64 bits, og nøkkelen er på 56 bits. Lengre klartekster blir behandlet i blokker på 64 bits. DES sin struktur er en variant av Feistel–strukturen som vi så ovenfor. Det er 16 runder av prosessering. Fra den originale 56 bits nøkkelen, genereres 16 subnøkler, en for hver runde. Dekryptering foregår på samme måte som kryptering (se Feistel–strukturen).

DES–algoritmen er kanskje den algoritmen som er/ har vært mest utbredt og er den algoritmen som det er brukt mest tid på å studere. Dersom vi ser nærmere på styrker og svakheter til DES er de hovedsakelig knyttet til: selve algoritmen og nøkkellengden. Gjennom årene har det vært forsøkt å utnytte eventuelle svakheter i algoritmen for å dekryptere meldinger som er generert ved hjelp av den. Så langt har ingen lykkes i å finne noen fatale svakheter i selve algoritmen [3]. Vi vil derfor se nærmere på nøkkellengden, kanskje kan vi finne noen svakheter der. Nøkkellengden til DES er 56 bit. Dette ble antydet som en svakhet allerede i starten av arbeidet med standardiseringen. I juli 1998 klarte en gruppe å knekke en DES kryptert melding. EFF (Electronic Frontier Foundation) bygde en spesiell maskin for å knekke nettopp meldinger kryptert med DES. De brukte mindre en tre dager på angrepet (EFF 1998). Dette viste at nøkkellengden var langt fra tilstrekkelig. Dersom det er slik at et angrep basert på "brute force" er den eneste muligheten, vil det være enkelt å øke nøkkelstørrelsen. Dersom vi øker nøkkelstørrelsen til 128 bit, vil en EFF "DES-knekker" bruke 10^8 år. Selv om vi kunne øke hastigheten med en faktor på 10^{12} vil det fortsatt ta over 1 millioner år å knekke koden (Stallings 2003).

3.3.2 Trippel DES

Trippel DES var svaret på trusselen fra kryptoanalytikerne. Siden vi har en god algoritme, og det er nøkkellengden som er problemet, ble DES gitt et nytt liv som 3DES (trippel DES). 3DES bruker tre nøkler og tre utførelser av DES algoritmen for å gi økt sikkerhet. Med tre slike distinkte nøkler har 3DES en effektiv nøkkellengde på 168 bit. 3DES som ble lansert i 1985, ble snart en ny standard for sikker meldingskryptering, og har stor utbredelse i dag [4].

3.4 AES – Advanced encryption standard

3DES har mange gode argumenter for videre bruk. Den bruker en sikker og gjennomtestet algoritme, og den har en nøkkellengde som er lang nok i overskuelig framtid. Det er allikevel noen problemer knyttet til 3DES. 3DES er treg dersom den implementeres i programvare,

[3] Ingen har offentlig gått ut med informasjon som kan tyde på dette. Hva NSA, KGB m.fl. har gjort er en annen sak

[4] 3DES ble gjort til en del av Data Encryption Standard i 1999. 3DES brukes bl.a. av USAs regjering og statsapparat

og den har en liten blokkstørrelse (56-bit) utifra sikkerhets og effektivitets hensyn. På bakgrunn av dette er ikke 3DES en god kandidat for en kryptoalgoritme som skal kunne brukes i lang tid fremover. Derfor utlyste NIST i 1997 bidrag til det som skulle bli AES (Advanced Encryption Standard). Det ble spesifisert at algoritmen skulle kunne bruke nøkler av lengde: 128, 192 og 256 bit. Og den skulle ha en blokklengde på 128 bit. Etter flere runder med mange gode forslag, ble algoritmen Rijndael [5] valgt i november 2001. Vi skal ikke gå i detaljer om denne algoritmen, men bare slå fast at så langt vi vet i dag er det en veldig sikker og god algoritme. Det kan være verd å merke seg at algoritmen er effektiv også i software noe som var viktig for NIST.

3.5 Asymmetrisk kryptografi

De symmetriske algoritmene forutsetter at for hver gang noe skal sendes kryptert mellom en sender og en mottaker, så har mottaker og sender først blitt enig om en hemmelig nøkkel som de deler. Dette er felles for alle de ulike asymmetriske algoritmene. Nøkkelen kan ikke sendes kryptert fordi vi ikke har noen nøkkel for å kryptere denne meldingen med. For å utveksle meldinger med asymmetrisk kryptografi er det derfor nødvendig med enten et møte mellom de kommuniserende partene, eller at de på forhånd har blitt enige om hvilke nøkler som skal brukes. Systemet vil heller ikke skalere godt, dersom det er 10 brukere som skal kommunisere sammen trenger de 45 nøkler, 100 brukere trenger 4950 ulike nøkler (Schneier 2000). Så selv om vi hadde sikre algoritmer for å kryptere meldinger mellom to parter, fantes det ikke noen gode løsninger på problemet med nøkkeldistribusjon. I (Schneier 1996) sammenlignes dette med at to mennesker roper tall til hverandre i en kafè fylt med matematikere, slik at når de er ferdige har de begge kommet fram til det samme tilfeldige tallet, uten at noen på kafèen vet det. Det høres umulig ut, men i 1976 publiserte Whitfield Diffie og Martin Hellman en artikkel som viste at nettopp det var mulig. Og det er dette som er bakgrunnen for det som også blir kalt offentlig nøkkel kryptografi eller asymmetrisk kryptering. Ideen går ut på å bruke en matematisk funksjon som er enkel å beregne en vei, men umulig å reversere, dvs. komme tilbake til det opprinnelige tallet. Det finnes flere matematiske funksjoner med slike egenskaper, blant annet faktorisering av store primtall, kvadratrot modulo et sammensatt tall m.fl. Asymmetrisk kryptering kan også benyttes til signering av digitale data.

3.5.1 Offentlig nøkkelkryptografisystemer

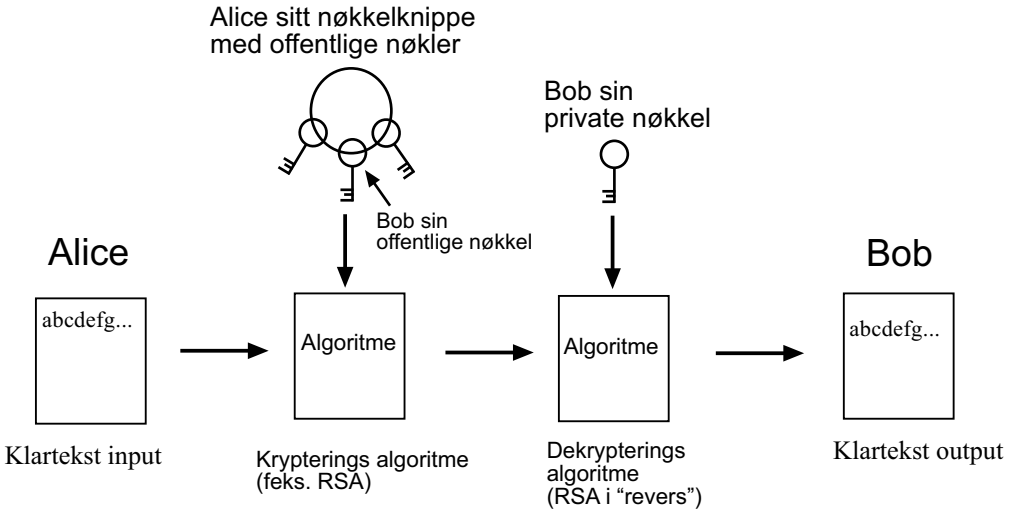
Et offentlig nøkkel system har seks ingredienser:

- Klartekst - lesbare data som skal behandles av algoritmen.
- Krypteringsalgoritme - Algoritme som benyttes på klarteksten for å produsere kryptert tekst.

[5] Oppkalt etter skaperne Dr. Joan Daemen og Dr. Vincent Rijmen fra Belgia

- Offentlig og privat nøkkel - Et par av nøkler hvor den ene benyttes for kryptering (offentlig) og den andre for dekryptering (privat).
- Kryptert tekst - Teksten som output fra krypteringsalgoritmen. Avhengig av klartekst og nøkkel brukt.
- Dekrypteringsalgoritme - Algoritme som tar inn den krypterte teksten og tilhørende nøkkel for å reprodusere den originale klarteksten.

Som navnet tilsier er den offentlige nøkkelen åpen og gjort kjent for andre som vil kommunisere med en gitt part. Den private nøkkelen må holdes hemmelig, og skal kun benyttes av eieren av den. De vanligste offentlig-nøkkelkryptografi algoritmene benytter seg av en offentlig nøkkel for kryptering og en annen relatert nøkkel for dekryptering. Når systemet er på plass kan to parter f.eks.. Alice og Bob kommunisere sikkert på følgende måte se figur 3.5.1.



Figur 3.5.1: Asymmetrisk kryptering

1. Alice og Bob genererer hvert sitt par av nøkler som skal brukes til kryptering og dekryptering av meldingene.
2. Alice og Bob plasserer sine offentlige nøkler i et offentlig register, eller publiserer sin offentlige nøkkel på andre måter. Den tilhørende private nøkkelen holdes hemmelig. De må også holde orden på offentlige nøkler de får fra andre kommuniserende parter. Vi kan tenke oss at de har hver sitt nøkkelknippe med de offentlige nøklene de har hentet inn.
3. Dersom Alice ønsker å sende en privat beskjed til Bob så bruker hun Bob sin offentlige nøkkel for å kryptere beskjeden.

4. Når Bob mottar beskjeden kan han dekryptere den med sin egen private nøkkel. Ingen andre parter, vennligsinnede eller inntrengere kan dekryptere beskjeden siden de ikke har Bob sin private nøkkel.

De offentlige nøkkelkryptografisystemene løser problemet med nøkkeldistribusjonen. Offentlig nøkkelkryptografi kan også gjøre andre ting enn å kryptere/dekryptere meldinger på denne måten. De kan også brukes til digitale signaturer. En sender vil da "signere" meldingen med sin private nøkkel, som ingen andre har tilgang til. En mottaker kan sjekke signaturen ved å benytte senderen sin offentlige nøkkel. Systemet kan også brukes til kun å utveksle en sesjonsnøkkel som senere brukes i en symmetrisk kryptering. Grunnen til at det ofte skjer skal vi snart se nærmere på. Ulike asymmetriske algoritmer kan brukes til alle, eller flere av disse funksjonene.

3.5.2 RSA

RSA er en algoritme for asymmetrisk kryptering og er kanskje den mest utbredte. RSA står for Rivest, Shamir, Adlemann etter dens oppfinnere. RSA sin kompleksitet og sikkerhet ligger i det faktum at det er vanskelig å faktorisere store tall. De offentlige og private nøklene er funksjoner av store primtall. Krypteringsalgoritmen skapte stor oppstandelse da den kom fordi den løste to store problemer: nøkkelfordeling og signering. Den var også sagt å skulle være ubrytelig. De første to påstandene var for så vidt riktige, men den siste var nok feil. Noe av kritikken som er rettet mot systemer slik som RSA er at selv om de løser nøkkeldistribusjonen, er vi fortsatt avhengig av å sikre oss at brukeren er akkurat den vi vil kommunisere med, og at det er riktig offentlig nøkkel som vi har assosiert med denne personen. Det viste seg også at selv om algoritmen fungerte godt og var relativt enkel å forklare matematisk sett, var den veldig treg i hardware og software. Det er snakk om en faktor 1000 for hardware implementasjoner, og en faktor 100 for software implementasjoner sammenlignet med for eksempel DES. Dette gjør at i de aller fleste tilfeller blir RSA, El-Gamal, Diffie-Hellman osv. brukt i s.k. *hybridsystemer*.

3.5.3 Hybridsystemer

Hybridsystemer er systemer som kombinerer den symmetriske krypteringen sin hastighet og sikkerhet, med asymmetriske kryptosystemers løsning på nøkkelfordelingen. Måten de brukes på er at f.eks.. Alice som vil kommunisere med Bob lager seg en sesjonsnøkkel for symmetrisk kryptering som hun bruker for å kryptere meldingen med. Hun krypterer så sesjonsnøkkelen med Bob sin offentlige nøkkel og sender den krypterte meldingen sammen med den krypterte nøkkelen til Bob. Når Bob mottar dette utfører han de samme operasjonene i revers. Han bruker sin egen private nøkkel for å dekryptere sesjonsnøkkelen som han igjen bruker for å dekryptere meldingen. Dette brukes f.eks.. i e-post kryptering som PGP, S/MIME m.fl, Web sikkerhet, TCP/IP sikkerhet, avlyttingssikre telefoner m.m. (Schneier 2000)

3.6 Meldingsautentisering og digitale signaturer

3.6.1 Enveis hashfunksjoner

En måte å sikre meldinger eller andre typer data fra å bli forandret av en uvedkommende uten vår viten, er bruk av meldingsautentisering. Meldingsautentisering foregår slik at vi har en så kalt enveis hashfunksjon som beregnes over meldingen eller dataene som vi ønsker å lage et "fingeravtrykk" av. En enveis hashfunksjon fungerer som et fingeravtrykk, den er en liten datadel som kan brukes til å identifisere en større datamengde, f.eks. en e-post, en programnedlasting eller en elektronisk bok. Funksjonene kalles enveis fordi de tilhører matematiske funksjoner som er enkle å beregne en vei. En hash funksjon er en funksjon som komprimerer et input av vilkårlig lengde til et resultat av fast lengde. Det er mulig å utflre en hash avdenne oppgaven sin elektroniske form, og få en hashverdi. Det som derimot er omtrent umulig er å skrive et dokument som hasher til en tilsvarende hashverdi, dersom dokumentet ikke er helt likt. Hashfunksjonen produserer et elektronisk fingeravtrykk for akkurat min oppgave. Personer som vil laste ned oppgaven kan derfor også laste ned hashen av oppgaven, de kan deretter selv ta en hash av oppgaven og sammenligne for å se om den er lik. Dersom den er lik, kan de være sikre på å sitte med akkurat min oppgave. Eksempler på slike offentlige tilgjengelige algoritmer er SHA-1, MD5 (på vei ut) og RIPEMD-160.

3.6.2 Digitale signaturer

Signaturer har i lang tid vært viktig ved kontraktsinngåelser og for å identifisere mennesker. Bruken av signaturer gir oss en rekke goder:

1. Signaturen er autentisk. Den knytter den som har signert et dokument til det signerte dokumentet.
2. Signaturen kan ikke forfalskes, den er bevis for at det virkelig er den som signerte dokumentet og ikke noen annen som går god for avtale, innhold m.m. i dokumentet.
3. Signaturen "hører til" dokumentet og kan ikke overføres til et annet dokument. Signaturen kan ikke gjenbrukes.
4. Dokumentet kan ikke forandres. Etter at dokumentet er signert skal det ikke kunne forandres på.
5. Signaturen kan ikke benektes. Signaturen og dokumentet er fysiske ting, den som signerte kan ikke i etterkant hevde å ikke ha gjort det.

Dette er idealiserte egenskaper til signaturer, i det virkelige livet er det ikke helt slik som i disse punktene. Det er allikevel slik at det er vanskelig å forfalske signaturer, og forbrytelser som sjekkforfalskning m.m. blir slått hardt ned på. Vi ønsker å kunne tilby flest mulig av disse egenskapene for brukere av datanettverk/internett. Digitale signaturer er viktige for

å kunne bruke nettet til handel, både mellom bedrifter og mellom bedrifter og kunder. Det er flere problemer knyttet til signaturer i datamaskinverdenen. Blant annet er det mulig å lage perfekte kopier av alle mulige typer data. Det er også relativt enkelt å gjøre forandringer i filer etter de er opprettet, uten at det oppdages. Det finnes flere måter å benytte de kryptografiske teknikkene vi har sett på for å gjenskape signaturer slik at de kan brukes også for elektroniske data. En av de mest vanlige er bruken av asymmetrisk kryptering for å signere meldinger. Dette kan gjøres på flere måter, den vanligste er å bruke nøklene for asymmetrisk kryptering i en annen rekkefølge. La oss tenke oss at Bob vil sende en melding til Alice, at andre kan lese meldingen er ikke så viktig for Bob. Han ønsker også at Alice skal være sikker på at det er Bob som har sendt meldingen og at den ikke er forandret på av uvedkommende. Bob kan gå fram på følgende måte:

- Bob krypterer meldingen med sin *private* nøkkel og sender den til Alice.
- Alice dekrypterer Bob sin melding ved hjelp av Bob sin *offentlige* nøkkel.

Dersom Alice og Bob har benyttet seg av en pålitelig tredjepart for å utveksle offentlige nøkler, kan Alice være sikker på at meldingen hun har fått er fra Bob. Det er nemlig ingen andre enn Bob som har mulighet til å kryptere en melding ved hjelp av Bob sin private nøkkel. Alle kan for så vidt dekryptere meldingen og finne ut at den kommer fra Bob, men ingen andre har mulighet til å kryptere den med Bob sin private nøkkel. Ofte er det viktigere å kunne bevise hvem meldingen kom fra enn å holde den hemmelig. I implementasjoner av et slikt system vil signaturen gjøres på en hash av meldingen, både av effektivitetshensyn og fordi det er sikrere. Spesielle digital-signatur algoritmer benytter seg av lignende systemer. Eksempler på slike algoritmer er DSS, RSA og ElGamal [6] og algoritmer som baserer seg på elliptisk kurve kryptografi [7] (Schneier 2000).

3.7 Autentiseringsprotokoller og kryptografi

Her skal vi kort si litt om forholdet mellom kryptering og autentiseringsprotokoller, uten å forklare protokollbegrepet nærmere. Protokoller kommer vi tilbake til i neste kapittel. Autentiseringsprotokoller bruker kryptografiske metoder for å autentisere Bob eller Alice over et datanettverk. Alle de kjente og klassiske autentiseringsprotokollene benytter seg av alle eller noen av teknikkene som er presentert i dette kapittelet. I et datanettverk hvor en eventuell inntrenger eller spion kan få tak i alle meldinger, gjøre forandringer på de, slette, samle inn, er vi avhengig av å sikre oss med hensyn til integritet, autentisering og konfidensialitet. Disse tre egenskapene som er så sentrale ved sikkerhet i nettverk kan alle sammen tilbys ved hjelp av de kryptografiske teknikkene vi har sett. I protokollene fungerer disse teknikkene sammen for å tilby de ulike egenskapene på forskjellige nivåer.

[6] RSA og ElGamal finnes både som algoritmer for asymmetrisk kryptering av sesjonsnøkler og som rene digital-signatur algoritmer

[7] Slike algoritmer er mer effektive i noen tilfeller

Det er her det lett kan snike seg inn feil og åpnes muligheter for angrep som kan være svært vanskelige å avdekke. De formelle metodene som benyttes finner logiske feil og designfeil ved de uformelle beskrivelsene av hva en protokoll skal gjøre. Å avdekke disse er et lite skritt på veien mot sikre protokoller. Vi vet at en ting er hvordan protokollen oppfører seg i teorien, en helt annen ting er hvordan den fungerer i praksis. Ofte kan en implementasjon virke god på papiret men i den virkelige verden er feilene og hullene mange. Ikke alle kan forklares utifra programmeringsfeil og bugs. Ofte har vi utilsiktede følger av implementasjonsmessige tilpasninger. Noen ganger åpner implementasjonen for buffer overskridelser og lekkasjer av data som kan utnyttes for å få informasjon om protokollen og slik finne angrepsmuligheter. Et eksempel på buffer-overskridelsesfeil kan være Microsoft sin feil i UPnP[8]. Som Bruce Schneier påpeker (Schneier 2002) er Microsofts feil en klar programvare-feil. Det er en "bug" som det er lett å teste for, og som skal fanges opp av en kvalitetssjekk av koden til programmet. Selv om vi tester en spesifikasjon, og ikke finner noen feil kan vi ikke slå oss til ro med det. Vi må kvalitetssikre hele prosessen fra uformell → formell spesifikasjon → implementasjon.

[8]Universal Plug and Play - "The UPnP architecture defines common protocols and procedures to guarantee interoperability among network-enabled PCs, appliances, and wireless devices" (sakset fra UPnP forum)

Kapittel 4

Introduksjon til protokoller

Før vi kan benytte oss av krypteringsfunksjonene som vi har sett på i kapittel 3, må vi finne ut mer om hva en protokoll egentlig er for noe. For å forstå autentiseringsprotokoller må vi både kunne noe om de kryptografiske teknikkene som ligger til grunn for å kunne utføre en autentisering og vi må forstå hva en protokoll er, og hvordan den brukes.

4.1 Hva er en Protokoll?

En protokoll kan forstås som en serie trinn, som involverer to eller flere parter, for å gjennomføre en bestemt oppgave (Schneier 1996). La oss begynne med å se nærmere på hva denne definisjonen inneholder.

- "en serie trinn" - med dette menes at protokollen består av en sekvens fra start til mål. Hvert trinn i protokollen må utføres etter hverandre og ingen trinn kan utføres før det foregående trinnet er ferdig utført (m.a.o.. sekvensielt).
- "involverer to eller flere parter" betyr at minst to parter (menneske eller maskin) må være med for å få til en gjennomføring av protokollen.
- "for å gjennomføre en oppgave" betyr at protokollen må oppnå noe, gjennomføre en bestemt ting. En protokoll som ikke gjør noe, er vel å se på som bortkastet tid og ressurser.

4.2 Formålet med protokoller

I det daglige livet finnes det protokoller for nesten alle ting vi gjør. De er uformelle og derfor er det ikke alltid vi tenker på dem som protokoller. De har utviklet seg over tid, alle følger dem, vet hvordan de utføres og de fungerer relativt godt. Eksempler på slike "hverdagsprotokoller" kan være: - hvordan en hilser på personer i ulike sammenhenger,det

kan være på jobb, eller privat. Hvordan telefonsamtaler utspiller seg. Hvordan en oppfører seg når en er på fotballkamp etc. I våre dager foregår mer og mer menneskelig interaksjon ved bruk av datanettverk istedenfor ved "ansikt til ansikt" kommunikasjon. En stor forskjell mellom mennesker og datamaskiner er at mennesker er veldig fleksible og kan tilpasse sin "protokoll" til nye situasjoner som ligner på situasjoner de har vært i tidligere. Tenk deg at du skal poste et brev. Du må skrive adressen på forsiden av konvolutten, sette på et frimerke øverst i høyre hjørne og putte det i en postkasse. Dette vil du som menneske også kunne gjøre dersom du er på ferie i utlandet, selv om frimerket ser litt annerledes ut og postkassene har andre farger osv. Så fleksible er ikke datamaskiner. Mange "ansikt til ansikt" protokoller bygger på at personene er tilstede for at de skal være rettferdige og sikre. Ville du f.eks.. sende penger med en vilt fremmed for å gå i butikken for deg? Eller ville du spilt kort med noen som ikke blander og deler ut kortene foran deg? Grunnen til at disse spørsmål blitt stilt er at (dessverre) er det naivt å tro at datanettverk er til å stole på. Det er til og med naivt å tro at de som vedlikeholder nettverkene er til å stole på. Selv de som designer og programmerer dataprogrammer er nødvendigvis ikke til å stole på. De fleste er det nok, men de få som ikke er til å stole på kan gjøre stor skade [1]. Ved å formalisere protokoller kan vi finne svakheter som uærlige personer prøver å utnytte. Slik kan vi utvikle protokoller som ikke kan utnyttes til egen fordel. I tillegg til å formalisere forløpet av handlinger kan vi ved hjelp av protokoller lage en abstraksjon mellom det som protokollen skal utføre og de mekanismer som implementerer selve protokollen. På den måten kan samme protokoll brukes i ulike systemer, og vi kan undersøke protokollen uten å måtte grave oss ned i implementasjonsdetaljene. Når vi er sikre på at vi har en god protokoll, kan vi implementere den i ulike systemer.

4.3 Karakteristikker for protokoller

Etter å ha sett på en definisjon av hva en protokoll kan være, skal vi se på noen av de karakteristikkene vi har kommet fram til en protokoll i dataverdenen bør ha. Slike karakteristikker gir oss et bilde av hva en protokoll bør inneholde og hva som ventes av partene som skal delta i utførelsen av protokollen.

- Alle som er involvert i en protokoll må kjenne protokollen og alle trinn i den på forhånd. Og alle må si seg enig i å følge den.
- Protokollen må være utvetydig, til enhver tid må alle parter være sikre på hva de skal gjøre. Trinnene må være veldefinerte og det skal ikke være rom for misforståelser.
- Protokollen må også være fullstendig, dvs. at det må finnes handlinger for alle mulige situasjoner.

[1] Dette med tillit er vanskelig. Mange selskaper anmelder ikke innbrudd i sine datanettverk, dels for ikke å få dårlig publisitet og for ikke å miste tillit fra sine kunder. (se Schneier (2000) side 391.

- Hvert trinn involverer minst to ting; 1. beregning utføres av en eller flere parter 2. beskjed sendes mellom noen av partene.

4.4 Eksempler på ulike typer protokoller

Vi har ulike typer av protokoller. Først vil jeg se nærmere på noen slike typer og så komme med noen enkle eksempler på protokoller. I del 4.4.1 ser vi noen viktige symboler som brukes i autentiserings og nøkkelutvekslingsprotokoller (Schneier 1996).

4.4.1 Noen navn som brukes i forbindelse med protokoller

I litteraturen om protokoller er det vanlig å bruke en del standard navn på parter og elementer som brukes i autentiserings og nøkkelutvekslingsprotokoller.

Symbol	Funksjon
Alice	part nummer 1
Bob	part nummer 2
Eve	lytter på kommunikasjonen (eavesdropper)
Mallory	ondskapsfull aktiv angriper
Trent	pålitelig tredjepart
Walter	vekter, passer på Alice og Bob i noen protokoller
Peggy	er den som beviser
Victor	verifiserer
E_A	Kryptering med en nøkkel som Trent deler med Alice
I	Indeksnummer
K	Nøkkel
N_a, eller I_a	Et tilfeldig tall, ofte kalt *nonce*

4.4.2 Protokoller med pålitelig tredjepart

En pålitelig tredjepart er en person uten interesse i saken som er sett på som pålitelig og sannferdig av begge de involverte parter. En slik pålitelig tredjepart kan hjelpe til å fullføre protokollen mellom to parter som gjensidig ikke har tillit til hverandre. Eksempler på slike personer i samfunnet er advokater som kan fungere som tredjeparter f.eks.. ved oppgjør i bo etter dødsfall o.l. Et annet eksempel kan være oppgjør ved kjøp og salg mellom to mennesker. La oss se på et eksempel. Dersom Alice og Bob har avtalt kjøp og salg av en bil, så vil ikke Alice gi fra seg bilen før hun har fått pengene. På samme måte vil ikke Bob betale før han har fått bilen. Dette kan løses ved en protokoll som fungerer på følgende måte.

1. Alice gir papirene på bilen til en advokat.

2. Bob gir en sjekk til Alice.
3. Alice går i banken og hever sjekken.
4. Etter å ha gitt sjekken et visst tidsrom for å godkjennes gir advokaten papirene på bilen til Bob. Dersom Alice ikke får utbetalt pengene fra sjekken vil hun si ifra til advokaten om dette, slik at Bob heller ikke får papirene på bilen.

I dette eksempelet stoler både Alice og Bob på at advokaten skal gjøre det protokollen sier. Advokaten vil utføre det som protokollen sier, det er jo det han får penger for. I dette tilfellet fungerer advokaten som en slags oppbevaring for en ikke-oppfylt kontrakt. (eng. escrow agent). Andre eksempler på slike pålitelige tredjeparter er banker, børser og sorenskrivere. Selve konseptet er gammelt og før i tiden fungerte embetsmenn på denne måten, det kunne være prester m.fl. Disse betrodde tredjepartene var mennesker med en spesiell sosial rolle og posisjon, men som ved å misbruke tilliten ville miste denne. Denne påliteligheten eksisterer ikke alltid i den virkelige verden, men er snarere et ideal å strekke seg etter. Er det mulig å finne igjen denne tilliten i dataverdenen, kan idealet overføres og kanskje virkeliggjøres der? Dersom vi prøver å virkeliggjøre dette i dataverdenen dukker det opp en del problemer med disse "virtuelle" pålitelige tredjepartene.

- å finne noen du kan stole på er enkelt dersom du vet hvem en person er og kan møte personen ansikt til ansikt. To parter som er mistenksomme overfor hverandre vil sannsynligvis også være det overfor en ansiktsløs tredjepart i et datanettverk.
- datanettverket må betale for kostnaden ved å utvikle en slik tredjepart. Hvem skal betale regningen? (tenk på hva en advokat tar betalt i timen)
- det vil oppstå forsinkelser ved enhver bruk av en slik protokoll
- tredjeparten må bistå i en mengde transaksjoner og kan derfor bli en flaskehals i systemet/nettverket. Å ha flere slike pålitelige tredjeparter vil løse noe av problemet, men vil igjen øke kostnadene.
- siden alle på nettverket bruker denne tredjeparten vil han representere et strategisk punkt dersom noen skulle angripe nettverket.

4.4.3 Protokoller med dommer

Fordi det er involvert store kostnader ved hele tiden å måtte benytte seg av en pålitelig tredjepart, bruker noen protokoller isteden en dommer som avgjør eventuelle tvister. En slik protokoll består av to subprotokoller. Den første subprotokollen er den som utføres hver gang partene ønsker å fullføre protokollen. Den andre subprotokollen utføres bare dersom det skulle oppstå en uenighet. Da vil en pålitelig tredjepart komme inn og avgjøre hvem som har retten på sin side. I denne protokollen kalles derfor den pålitelige tredjeparten for en dommer. Dommeren fungerer omtrent som den pålitelige tredjeparten, men deltar

ikke i alle gjennomførelser av protokollen. Dommeren kalles kun inn dersom det oppstår en uenighet. Vi finner slike dommere i f.eks.. rettsapparatet. Alice og Bob kan bli enige om en kontrakt uten en dommer. Dommeren ser ikke kontrakten før Alice eller Bob eventuelt går til rettssak.

Vi kan formalisere denne protokollen som to subprotokoller.
Første subprotokoll

- (1) Alice og Bob forhandler fram en kontrakt.
- (2) Alice signerer kontrakten.
- (3) Bob signerer kontrakten.

Andre subprotokoll med dommer. (utføres dersom det oppstår en uoverensstemmelse)

- (4) Alice og Bob fremstilles for dommeren.
- (5) Alice presenterer sine bevis.
- (6) Bob presenterer sine bevis.
- (7) Dommeren avsier sin dom på bakgrunn av bevismaterialet.

Forskjellen mellom en pålitelig tredjepart som i den første protokollen vi har sett på og dommeren i denne, er at dommeren ikke alltid er nødvendig. Faktisk er det slik at de aller fleste utførelser av protokollen gjøres helt uten at dommeren er inne i bildet. Hvis det ikke oppstår noen uenighet er bruken av en dommer unødvendig. Slike protokoller finnes både i den virkelige verden og på datanettverk. Protokollen hviler på de involverte partenes ærlighet. Dersom de skulle bli uenig finnes det data for å avgjøre hvem som eventuelt jukset. I en god protokoll av denne typen vil man også kunne finne identiteten til personen som jukset. I stedet for å forhindre juks er denne protokollen laget for å oppdage juksing. Det at juks oppdages fungerer preventivt og vil avskrekke juksing.

4.4.4 Selvregulerende protokoller

I tillegg til de to protokolltypene vi har nevnt over finnes det s.k. selvregulerende protokoller. De selvregulerende protokollene sikrer en rettferdig gjennomførelse. Det behøves ingen pålitelig tredjepart for å avgjøre tvister. Protokollene fungerer slik at dersom noen prøver å jukse vil protokollen oppdage dette og stoppe utførelsen. I en perfekt verden ville alle protokoller fungere på denne måten. Dessverre finnes det ikke slike protokoller for alle situasjoner.

4.5 Internettprotokoller

Med Internettprotokoller tenker vi på alle de protokollene som brukes ved kommunikasjon, vedlikehold, statusrapportering osv. på nettet. Vi har de underliggende nettverksprotokollene, IP og TCP/UDP. I tillegg finnes HTTP, HTTPS (sikker http), SSH (secure login connection) og dens avarter rsh, rlogin, rcp osv. som ligger over TCP/IP igjen. Disse protokollene definerer helt grunnleggende elementer som adressering, mottagelse og sending av datapakker, hvordan formatet på disse er osv. Vi har spesialiserte protokoller for å gjøre bestemte ting. For eksempel så er SNMP (Simple Network Managment Protocol) en protokoll som kan formidle beskjeder om trafikken på punkter i nettverket, den tar seg av vedlikeholdsrapportering osv. IPsec er en protokoll som "ligger utenpå" IP og/eller TCP protokollen for å tilby autentisering og/eller konfidensialitet.

4.6 Sikkerhetsprotokoller

Sikkerhetsprotokoller også kjent som kryptografiske protokoller [2] er bare en bit i puslespillet for å oppnå sikkerhet i et nettverk. I tillegg trenger vi mekanismer og protokoller for å løse problemer utover selve krypteringen/dekrypteringen. Det kan være å *autentisere* partene i kommunikasjonen, teknikker for å sikre *integriteten* til meldinger, og måter å løse problemet med distribusjonen av offentlige nøkler i forbindelse med kryptering/dekrypteringen. En viktig del av sikkerhetsprotokollene er krypteringen/dekrypteringen slik vi finner det i kapittel 3.

4.6.1 Hva ønsker en å oppnå med en sikkerhetsprotokoll

Vi ønsker hemmeligholdelse på et visst nivå, å sikre oss mot at andre kan "avlytte" kommunikasjonen. Vi vil ansvarliggjøre partene i en samtale, forsikre oss om at deltagerne er de som de utgir seg for å være. Siden vi i en kommunikasjonssituasjon kan ønske oss begge disse tjenestene, er det ofte slik at vi trenger flere ulike typer protokoller for å få tilgang til de. Dersom vi ønsker å sende en e-post, må vi først logge oss på maskinen, da er det maskinen som prøver å autentisere hvem personen som logger seg på er, ofte ved bruk av passord. Denne autentiseringen gjør datanettverket i stand til å gi oss de nødvendige nettverksressurser og tilgang til egne filer, e-postprogram osv. På samme måte er det når vi starter e-post programmet, vi blir bedt om å autentisere oss selv, og e-postprogrammet henter ned ny e-post til oss. Allerede før vi har satt i gang med å skrive e-posten vi skal sende, har vi vært igjennom minst to ulike autentiseringsprotokoller. Vi logger oss på nettverket og vi autentiserer oss overfor e-post serveren. Innlogging på en maskin benytter seg av Kerberos i Windows, Unix bruker ssh til innlogging på andre maskiner, e-post programmer bruker ofte en av; Kerberos, SMTP og SSL.

[2] Kryptografisk protokoll – protokoll som benytter seg av kryptografi.

4.7 Autentiseringsprotokoller

Autentiseringsprotokollene er protokoller som er laget spesielt for at to eller flere parter skal kunne være sikre på identiteten til parten(e) som det kommuniseres med. I et distribuert datasystem er det nødvendig med mekanismer som gjør en agent på nettverket i stand til å være sikker på en annen agent sin identitet. En agent må være sikker på at han virkelig kommuniserer med valgt agent, snarere enn med en som gir seg ut for å være agenten. Dette er rollen som en autentiseringsprotokollen skal ta seg av. Generelt har vi en *initiator* A som ønsker å etablere en sesjon med en *responder* B, kanskje ved hjelp av en server *S*. Når protokollen er gjennomført skal responder være sikker på initiators identitet og vice versa (Lowe 1997). De fleste slike protokoller bygger på en delt hemmelighet, og protokollen blir ofte utført ved hjelp av en pålitelig tredjepart. En slik autentisering kan gjøres ved hjelp av et passord, noe vi mennesker har brukt i uminnelige tider. Et godt eksempel kan være historien om Ali Baba og hans røvere som bodde i en hule. Huleinngangen var dekket av en stein og den kunne bare åpnes med de magiske ordene: "Sesam, Sesam lukk deg opp". I dag bruker vi fortsatt dette systemet kombinert med et brukernavn i de aller fleste datasystemer. Disse systemene bruker også andre former for autentisering slik som smartkort, biometri m.m. som vi ikke skal komme inn på i denne oppgaven. Generelt kan vi si at autentisering baserer seg på noe vi vet (f.eks.. passord), noe vi har (f.eks.. id-kort) eller noe vi er (f.eks.. fingeravtrykk). De autentiseringsprotokollene vi skal se nærmere på baserer seg ofte på en "Challenge - Response" mekanisme. I forbindelse med autentisering ved hjelp av passord er vi avhengige av at den vi skal autentisere oss mot allerede har dette passordet, for å kunne sjekke om vi har riktig passord. I den protokollen vi skal se nærmere på i kapittel 7 utføres en autentisering uten noen slik form for tidligere utveksling av passord o.l.

4.8 Sammendrag

Vi har i dette kapittelet sett nærmere på protokoller. Vi har prøvd å forklare med dagligdagse eksempler hva en protokoll er, og vi har sett på noen ulike typer av protokoller. Til slutt kom vi inn på autentiseringsprotokoller som er et av hovedtemaene i denne oppgaven.

Kapittel 5

Feil ved og angrep på protokoller

I dette kapittelet skal vi se på de feil og angrep protokoller kan være utsatt for. Feilene og angrepene kan ramme alle typer protokoller, men vi skal spesielt se på de som rammer autentiseringsprotokoller noe som er viktig i denne oppgaven. For å forstå hvor viktig arbeidet med formelle metoder er, i forhold til autentiseringsprotokoller, må vi se hvor enkelt slike feil kan snike seg inn og hvor smarte angrepene kanvæe. Vi skal se nærmere på de formelle metodene i kapittel 6, 8og 9.

5.1 Ulike typer feil

I denne delen skal vi se på de feil som ofte går igjen og de angrep som rettes mot protokoller. I Carlsen 1994 har vi en oversikt over de ulike typer feil som kan opptre i en autentiseringsprotokoll og kan se hvordan de åpner for spesifikke angrep. De tre hovedkategoriene er:

I Funksjonelle spesifikasjonsfeil

II Implementasjonsavhengige feil

III Implementasjonsfeil.

5.1.1 Funksjonelle spesifikasjonsfeil

Feil og mangler i den formelle spesifikasjonen vil naturlig lede til feil i en implementasjon av spesifikasjonen. For å unngå slike feil er det viktig at en i den formelle beskrivelsen av protokollen tenker nøye gjennom hva en ønsker å oppnå med protokollen, og at det finnes verktøy for å avdekke feil ved en slik spesifikasjon. Vi skal komme tilbake til hvordan slike feil arter seg.

5.1.2 Implementasjonsavhengige feil

Implementasjonsavhengige feil opptrer når vi har en korrekt formell spesifikasjon som kan lede til både riktige og feilaktige implementasjoner. Disse feilene opptrer som følge av en ufullstendig, snarere enn en direkte feilaktig formell spesifikasjon. Denne typen feil og mangler kan være vanskelig å finne. Problemet ligger da ofte i hvor detaljrikt protokollspesifikasjonsspråket som brukes er. Vi vil både ha et språk som sikrer oss mot mangler på grunn av for liten uttrykksfullhet, men samtidig bør det ikke være for komplisert. Et språk som er for detaljrikt vil være vanskelig både å bruke og forstå.

5.1.3 Implementasjonsfeil

En implementasjonsfeil oppatår når vi har en feilaktig implementasjon av en riktig formell spesifikasjon. I denne anledning etterlyses verktøy og formelle metoder for å kunne verifisere at en implementasjon er riktig i forhold til en formell spesifikasjon. Et slikt verktøy kan være abstrakte tilstandsmaskiner, s.k. ASMer. De skal vi komme tilbake til i kapittel 8. Det som er vanskelig i protokollverifikasjon, er å finne hvilke feil og hvilke angrep som kan ramme en protokoll. Når feilen først er funnet, eller et vellykket angrep er gjort, er det enkelt å verifisere hvorvidt feilen finnes eller om angrepet er vellykket.

5.2 Angrep

En protokoll kan angripes på flere forskjellige måter. Når det gjelder kryptografiske protokoller er det i hovedsak tre måter å angripe disse på (Schneier 1996). Enten kan man angripe 1. de kryptografiske algoritmene, 2. de kryptografiske teknikkene eller 3. selve protokollen. Siden jeg går utifra at de kryptografiske algoritmene og teknikkene fungerer slik de skal, vil vi se nærmere på angrep rettet mot selve protokollen. De fleste angrep kan grovt deles inn i to ulike typer, vi har *aktive* og *passive* angrep.

5.2.1 Passive angrep

De passive angrepene består i at en angripende part prøver å tilegne seg informasjon ved å smuglytte på kommunikasjonen som foregår mellom de to partene som er involvert i protokollen. Slike angrep er vanskelig å oppdage, og protokollen bør derfor være konstruert slik at den motstår de. De passive angriperne er vanskelig å oppdage fordi alt de gjør er å lytte på trafikken.

5.2.2 Aktive angrep

De aktive angrepene utføres av en angriper som aktivt prøver å gjøre forandringer i utførelsen av en protokoll til sin egen fordel. Det kan være for å utgi seg for å være en av partene

i en kommunikasjon, gjøre forandringer på meldinger, slette meldinger, gjenta gamle meldinger eller gjøre forandringer lokalt på maskinen til en av partene. Selv for relativt "enkle protokoller" som kun tar for seg autentisering kan slike angrep være vanskelig å gardere seg mot. De passive angrepene prøver å tilegne seg informasjon gjennom analyse av de meldinger som de kan få tak i. Aktive angrep kan ha flere mål, de er kraftigere og kan derfor oppnå flere ting. Angriperen kan være interessert i :

- Tilgang til hemmelig eller sikret informasjon
- Bruk av systemressurser
- Tilgjengelighetsangrep (Denial Of Service Attacks)
- Å ødelegge eller endre informasjon
- Uautorisert tilgang til systemet

Aktive angrep er mer alvorlig enn passive for de som er involvert, særlig dersom partene ikke stoler på hverandre. Angriperen trenger heller ikke være en fra "utsiden" av systemet. Det kan være en som har tilgang til systemet eller det kan være en systemansvarlig, og nettopp disse utfordringene er det som gjør det å lage gode og sikre protokoller så vanskelig. Dersom angriperen er en av partene involvert i protokollen vil vi kalle den for en "juksemaker". Den som jukser kan enten være passiv; i det den som jukser bare er ute etter informasjon, eller aktiv; i det han/hun prøver å gjøre forandringer i selve protokollgjennomføringen. Å lage protokoller som er robuste mot uærlige parter er veldig vanskelig, men det kan være mulig å konstruere protokollen slik at den andre parten blir gjort oppmerksom på det, dersom den andre parten jukser. Å lage protokoller som sikrer mot aktive juksemakere er vanskelig, det som derimot må være et minstekrav er at protokollen sikres mot aktive angripere, vi kan jo tenke oss at systemet forøvrig er konstruert slik at utenforstående ikke har mulighet til å få tilgang til noen systemressurser.

5.3 Typer av angrep

En forståelse av de ulike angrep som en protokoll kan utsettes for er nødvendig i arbeidet med design, testing og forbedring av protokoller. Vi trenger kunnskap om hvordan angrepene kan utføres og forståelse av hvordan en eventuell angriper vil tenke og handle. For å analysere kryptografiske protokoller er det derfor nyttig med en oversikt over de ulike former eventuelle angrep kan ha. Hva rettes angrepene mot i protokollen? En slik oversikt kan hjelpe oss i å se hvilke svakheter som finnes i ulike protokoller. Det å kunne forstå en angriper og de mål han/hun har er viktig for å finne de svakheter en gitt protokoll nødvendigvis har. Det er selvfølgelig flere måter å nærme seg design av protokoller. En av måtene kan være å lage seg designprinsipper som en følger i utvikling og implementasjon av en protokoll. Uansett ser det ut til at det å ha en god forståelse av angripere og de

ulike angrep som finnes er nødvendig i arbeidet med en ny protokoll. Det er flere måter å karakterisere de ulike angrepene på. Vi har sett hvordan de grovt kan deles inn i aktive og passive angrep, en mer detaljert taksonomi over feil og tilhørende angrep finner vi i Carlsen (1994) og en lignende men utvidet oversikt i Gritzalis og Spinellis (1997). En litt annen vinkling er å karakterisere angrepene utifra inntrengerens mål og roller slik Xu, Kedem, og Gong (2000) har gjort.

5.3.1 Taksonomi over angrep

I følge Carlsen (1994) og Gritzalis og Spinellis (1997) kan en detaljert taksonomi se ut som følger:

1. Elementære protokollfeil
2. Passord/nøkkelgjettingsfeil
3. Gjentakelses angrep
4. Parallelle sesjoner
5. Interne protokollfeil
6. Kryptosystemfeil

5.3.2 Elementære protokollfeil

De elementære feilene i en protokoll kan være feil som skyldes mangler i selve protokollformaliseringen. Disse feilene gjør protokollen veldig enkel å angripe. Et eksempel fra (Carlsen 1994) er bruk av signaturer og krypteringen av disse. Et annet eksempel er at i enkelte systemer foregår autentisering ved hjelp av brukernavn og passord. I den mye brukte protokollen *telnet* sendes både brukernavn og passord i "klartekst". Både brukernavn og passord kan derfor lett "snappes opp" av en bruker på nettverket hvor nettverkstrafikken passerer. Brukeren kan f.eks. utføre en "tcpdump" [1] for så å lese både brukernavn og passord direkte fra pakkene i nettverkstrafikken.

5.3.3 Passord/nøkkelgjettingsfeil

De feil og angrep som tilhører denne kategorien opptrer på grunn av den begrensede mengde passord som brukere rundt om på nettverket velger for å autentisere seg, generere nøkler osv.(Gong 1990). Siden brukere ofte velger passord fra et begrenset utvalg er det mulig med s.k. ordboksangrep. Det vil si at man bruker en ordbok over vanlige ord for det aktuelle språket og automatisk prøver om noen av de passer. For å kunne gjøre det er

[1] `man tcpdump` gir: `prints out the headers of packets on a network interface that match the boolean expression.`

det nødvendig for en angriper å få tak i enten passordfilen, eller mer vanlig en kryptert passordfil. Grunnen til at angriperen må være i besittelse av selve passordfilen i et eller annet format er at ofte vil et system ha en form for passordpolicy. Passordfilen sikrer at en og samme bruker bare kan skrive feil passord et maksimalt antall ganger, før kontoen til denne brukeren blir sperret og han/hun må få seg et nytt passord. Dersom noen får tak i selve passordfilen omgås denne passordpolicyen og det er mulig å teste passord mot filen så mange ganger man ønsker. Da kan et ordboksangrep være meget effektivt. Dersom en passordfil kommer på avveie er det nødvendig med personlig oppmøte for å få nytt passord. Uten personlig oppmøte kan en ikke være sikker på identiteten til brukeren som skal bytte til nytt passord[2] se også figur 5.3.1.

5.3.4 Gjentakelsesangrep

Disse angrepene baserer seg på meldinger fra tidligere protokollgjennomføringer som blir gjentatt av en angriper. Problemet opptrer vanligvis i protokoller hvor det ikke er gitt tidsstempler som kan si noe om hvor fersk en melding er. En angriper som benytter seg av et slikt angrep trenger ikke nødvendigvis vite hva som er inneholdt i en melding. Det kan godt være en melding eller meldingsdel som er kryptert, men som angriperen kan bruke i en ny kjøring av protokollen for å få tak i en sesjonsnøkkel eller for å autentisere seg overfor en annen part. Et godt eksempel på et slikt angrep kan vi finne i Needham-Schroeder protokollen (se kap. 7). Det klassiske eksempelet er basert på gjensendelse (replay) av kompromitterte kommunikasjonsnøkler og ble funnet i av Denning og Sacco (1981), tre år etter at protokollen ble presentert. Vi skal se nærmere på dette angrepet i del 7.3.2.

5.3.5 Parallelle sesjoner

Angrep som baserer seg på gjennomføring av parallelle sesjoner av en protokoll kalles også orakelbaserte angrep. Angriperen får tak i ønsket informasjon ved å utveksle passende protokollmeldinger. Vanligvis deles angrep som baserer seg på parallelle sesjoner eller orakler inn i to former: 1. Enkel–rolle og 2. Multi–rolle angrep (Snekkenes 1992). Forskjellen ligger i relasjonen mellom en part og rollen som denne parten har i protokollen. En enkel–rolle protokoll er definert som en protokoll hvor det er et en–til–en forhold mellom en part og rollen parten spiller i protokollen. I en multi–rolle protokoll er denne relasjonen en–til–mange. Vi tar med ett eksempel på en slik enkel–rolle feil i Shamirs tre–veis–protokoll (Schneier 1996). Protokollen skal muliggjøre sikker kommunikasjon mellom Alice og Bob uten at de på forhånd må utveksle verken hemmelige eller offentlige nøkler. Forutsetningen er en symmetrisk kryptering som også er kommutativ:

[2]Etter et innbrudd på Universitet i Oslo sine datamaskiner 14. november 2002, måtte alle brukere av universitetes datanettverk møte opp personlig for å få nytt passord. Bakgrunnen for dette var at uvedkommende hadde fått tilgang til passordfilen for alle brukere som var beskyttet av svak kryptering. Det gjorde at uvedkommende lett kunne utføre et ordboksangrep og få tak i mange gyldige brukernavn og passord. Se figur 5.3.1.

Figur 5.3.1: UiO nettverket hacket.

$E_A(E_B(P)) = E_B(E_A(P))$
Alice sin hemmelige nøkkel er K_a og Bobs er K_b, og de skal utveksle meldingen M. Her er protokollen:

1. A $\rightarrow$ B : $\{M\}_{K_a}$
2. B $\rightarrow$ A : $\{\{M\}_{K_a}\}_{K_b}$
3. A $\rightarrow$ B : $\{M\}_{K_b}$

Dersom protokollen opererer i et enkel–rolle miljø vil en bestemt part, for eksempel Alice bare kunne spille rollen A når den er gitt henne. Og på samme måte kan Bob kun ha rollen B. Dette vil legge noen begrensninger på hva en eventuell angriper kan gjøre. La oss se hvordan en angriper, Eve, kan bruke Alice som orakel for å få tilgang til den hemmelige meldingen M. Alice som er den som starter selve protokollgjennomføringen sender en forespørsel til Bob (melding 1). Denne meldingen fanges opp av Eve, og i steg 2 og 3 vil Eve utgi seg for å være Bob for å kunne få tak i den hemmelige meldingen M. Alice følger protokollen og vet ikke at hun avslører hemmeligheten til noen.

1. Alice $\rightarrow$ Eve_{Bob} : $\{M\}_{K_a}$
2. Eve_{Bob} $\rightarrow$ Alice : $\{M\}_{K_a}$
3. Alice $\rightarrow$ Eve_{Bob} : M

Angrepet kan unngås dersom A kan skille mellom forskjellige meldingstyper, dvs. skille mellom krypterte og ikke–krypterte meldinger. Da ville A i steg 3 ha en begrensning på at det kun er meldinger av type *kryptert* som skal sendes.

5.3.6 Interne protokollfeil

Interne protokollfeil oppstår som følge av at en av partene ikke gjør de nødvendige handlinger som protokollen krever. Et eksempel fra Carlsen (1994) er i steg 3 av Shamirs tre-veis-protokoll (over), hvor A må sjekke om meldingen som sendes er kryptert. Kravet om at meldingen skal være kryptert er viktig å få med eksplisitt i en spesifikasjon av protokollen slik at en implementasjon blir riktig.

5.3.7 Krypteringssystemfeil

Krypteringssystemfeil er feil som vi finner fordi en protokollspesifikasjon ikke alltid er designet spesielt for en krypteringsalgoritme eller system. Vi kan derfor få uheldige kombinasjoner mellom krypteringssystem og protokollspesifikasjon som kan føre til feil og åpninger for angrep. Slike feil er beskrevet av mellom andre Massey (1988). I Massey sitt eksempel brukes XOR funksjonen for å kryptere meldinger i Shamirs tre-veis protokoll. Det vil si

at en som kan se meldingene på nettverket kan da finne klartekstmeldingen ved å addere sammen de to klartekstmeldingene som A og B utveksler. De feil som oppstår i skjæringspunktet mellom krypteringsprotokoll og system er vanskelig å avdekke med generelle metoder og verktøy, de må derfor finnes og utbedres fra sak til sak [3].

5.4 Angriperens våpen og ressurser

Det er flere måter å se på en inntrenger i et nettverk. Hva kan en angriper gjøre? Hvilke ressurser har han til rådighet? Dette er forutsetninger som er viktige når vi skal designe og/eller spesifisere en autentiseringsprotokoll. Dolev og Yao har vist i sin artikkel "On the security of public key protocols" Dolev og Yao (1983), hvordan en kan modellere en inntrenger. Dolev og Yao tenker seg at en inntrenger kan manipulere meldingene som passerer gjennom systemet, noe som ikke er så urealistisk når en ser hvilke verktøy som finnes for å lese og gjøre forandringer på TCP/IP pakker i et nettverk. Angriperen kan slette, sende på nytt, forfalske og videresende meldinger, kun begrenset av kryptografiske føringer. Vi forutsetter at krypteringsalgoritmene i seg selv motstår angrep. En inntrenger forventes å kunne benytte de kryptoalgoritmer som andre agenter på nettverket har tilgang til, men har ikke tilgang til de nøkler som er benyttet av andre, bortsett fra offentlige nøkler. Det er senere vist at en s.k *Macchiavelli inntrenger* kan utføre alle de angrep som en *Dolev-Yao inntrenger* kan utføre Syverson, Meadows, og Cervesato (2000). Det at Macchiavelli inntrengeren kan gjøre det samme som Dolev-Yao inntrengeren, selv om den er svakere i det at den har færre muligheter for handlinger, er nyttig fordi det gjør en slik inntrenger enklere å modellere.

5.5 Sammendrag

Vi har sett på de ulike feil og angrep som kan rettes mot en protokoll. Det er forsøkt å gi en kategorisering av de, og noen eksempler for de ulike kategoriene er tatt med. Angrepene og feilene utnytter ofte spissfindige svakheter i protokollene. Dette gir oss store utfordringer når det gjelder design og verifisering av protokoller.

[3] Se Massey (1988) for en mer detaljert beskrivelse av hvordan utnyttelsen av XOR funksjonen foregår.

Kapittel 6

Formelle metoder og autentiseringsprotokoller

6.1 Innledning

Det å lage nye protokoller for autentisering kan virke som en enkel oppgave. De inneholder ofte bare noen få beskjeder og parter. Dessverre viser det seg å ikke være slik. Design av en autentiseringsprotkoll begynner med en spesifikasjon av hvilke krav protokollen skal oppfylle. Den som skal designe protokollen bør ha en klar oppfatning om hva protokollen skal utføre, og hva korrekthet med hensyn til protokollen skal være. Å uttrykke hva som gjør protokollen korrekt er en komplisert oppgave. Det er gjort flere forsøk på å finne uformelle krav til hva korrekthet i en autentiseringsprotokoll skal være (Whitfield Diffie 1992). Det kan være krav av typen "Sesjonsnøkler skal være hemmelige, og A(lice) og B(ob) skal ha mottatt akkurat de meldingene motparten sendte." En annen tilnærming er å gi spesielle krav til spesielle klasser av protokoller (Syverson og Meadows 1993). Å gi en formell beskrivelse av en protokoll kan være en god hjelp i å forstå problemene en designer av protokoller står overfor. Det kan også være med å "tette" abstraksjons-gapet mellom den uformelle forståelsen av korrekthet for protokollen, og den formelle protokollbeskrivelsen. Anvendelsene av formelle metoder har stort sett vært rettet mot å vise korrekthet i eksisterende protokoller, og finne feil og muligheter for angrep i disse. Det kan også være interessant å bruke formelle metoder i design av autentiseringsprotokoller. Det kan lede til enklere og raskere utvikling, og spare tid og penger på redesign av protokoller slik det ofte fungerer i dag. Det er ikke gjort mye forskning innenfor dette feltet, det finnes imidlertid uformelle retningslinjer for design av kryptografiske protokoller (Abadi og Needham 1996). En annen faktor som gjør arbeidet vanskelig er det som Roger Needham skriver om i sin artikkel "Programming Satan's computer" (Anderson og Needham 1995). I arbeidet med autentiseringsprotokoller må vi ta i betraktning at vi har å gjøre med en motstander som prøver å gjøre alt han kan for å utnytte svakheter i protokollen. En tredje og veldig interessant metode er å starte med en høynivå abstraksjon av protokollen som skal utvikles og

legge til flere og flere detaljer til denne etter hvert. Når høynivå spesifikasjonen er testet, kan det gjøres forbedringer og legges til nye nivåer (implementasjoner) av laget over. Slik vil en til slutt stå igjen med en detaljert spesifikasjon/implementasjon av protokollen som skal designes (Buttyan 1999).

Det har vist seg at selv om protokollene er forholdsvis enkle i størrelse og hva de skal utføre, sniker det seg inn små feil og åpninger for angrep som det kan ta lang tid å avdekke (se f.eks. del 7.2 i kapittel 7). Disse angrepene viser at det er behov for systematiske verktøy for å avdekke feil og mangler.

Hovedsakelig er de formelle metodene å finne i noen få hovedkategorier. I følge Meadows (1994) finner vi de formelle metodene i følgende kategorier:

1. Modellering og verifisering av protokoller ved hjelp av spesifiseringsspråk og verktøy ikke spesielt utviklet for autentiseringsprotokoller. Se 6.2.
2. Utvikling av ekspertsystemer som en protokolldesigner kan bruke i testing og verifikasjon av protokoller. Se 6.3.
3. Modellering og verifikasjon av protokoller ved hjelp av modallogikker. Baserer seg ofte på logikker for "knowledge and belief", epistemiske og doxiske logikker. Se 6.5.
4. Formelle metoder basert på omskrivingslogikker og algebraiske systemer. Se 6.6.

Vi skal her se nærmere på disse uten å gå i detaljer om alle de forskjellige tilnærmingene.

6.2 Bruk av generelle metoder

Her ser vi på autentiseringsprotokoll som en hvilken som helst annen protokoll eller program, og bruker verktøy og spesifiseringsspråk som er utviklet for generell bruk. Fordelene med disse metodene er at vi har godt utviklede verktøy som også etterhvert har mulighet for verifikasjon og automatisering. En kritikk av denne tilnærmingen kan være at de kan vise korrekthet, men ikke nødvendigvis sikkerhet (Rubin og Honeyman 1993). Målet med å bruke formelle metoder i arbeidet med autentiseringsprotokoller er ofte todelt:

1. Vi ønsker å verifisere formelt at en autentiseringsprotokoll tilfredstiller gitte sikkerhetskrav.
2. Vi ønsker å finne svakheter og feil i spesifikasjonen.

Vi har flere eksempler på slike tilnærminger; D.P. Sidhu foreslår å spesifisere protokollen som skal studeres, som en rettet graf. Varadharajan bruker også denne tilnærmingen. Senere har Varadharajan benyttet seg av LOTOS (Language of Temporal Ordering Specification)

for spesifikasjon av autentiseringsprotokoller (Rubin og Honeyman 1993). Andre eksempler er bruk av Ina Jo[1].

Sidhu og Varadharajan har uavhengig av hverandre benyttet endelige tilstandsmaskiner. De benytter seg av rettede grafer for å beskrive hver enkelt part. Slik blir det laget et tilstandsdiagram for hver enkel aktør i protokollen. De settes sammen for å gi en ikke-deterministisk endelig tilstandsmaskin. Den endelige tilstandsmaskinen kan så undersøkes ved hjelp av så kalt "reachability-analysis" som har som mål å finne eventuelle "deadlocks" ved å søke igjennom alle tilstandene til maskinen. Denne metoden er effektiv for å se om protokollen er korrekt i forhold til spesifikasjonen. Den tar derimot ikke for seg hva en eventuell ondsinnet angriper kan finne på.

Vi har også eksempler på bruk av abstrakte tilstandsmaskiner (ASM [2]) for spesifikasjon m.a. av Kerberos (Bella og Riccobene 1997). Det skal vi se nærmere på i kapitlene om ASM (8) og AsmL (9)[3]. Fargede Petri nett er også benyttet for å formelt modellere og analysere protokoller. En mer moderne tilnærming ser på partene i en protokollgjennomføring som Communication Sequential Processes (CSP) og bruker en s.k. Failure Divergences Refinement sjekker (FDR) for å teste modellen. Det er også de som har brukt høyereordens logikk (HOL) for å se nærmere på hvordan en kan modellere og bevise egenskaper til kryptografiske protokoller (Snekkenes 1992). Verktøy som fungerer sammen med HOL er også utviklet, bl.a. *Isabelle*[4] og *Convince*.

Problemet med slike generelle tilnærminger kan være at de ikke behandler spesielle sikkerhetsrelaterte egenskaper like godt som systemer som er spesialutviklet for bruk på sikkerhetsprotokoller. Søking i tilstandsrom, som mange av modellsjekkerene benytter seg av, kan være et problem da tilstandsrommet vokser eksponensielt.

6.3 Ekspertsystemer

Ekspertsystemer benyttes av protokolldesignere for å studere en spesiell protokoll. Disse systemene starter i en "ugyldig" tilstand og prøver å finne ut om det er mulig å finne frem til denne tilstanden fra en starttilstand. Denne tilnærmingen er fin til å finne svakheter i protokoller, men gir som de mer generelle tilnærmingene heller ikke noen garanti for sikkerheten. De fungerer godt dersom de brukes for å avdekke eksisterende feil, men det er lite sannsynlig at de finner nye ukjente feil. Vi skal se kort på noen av de mest kjente verktøyene beskrevet i Buttyan (1999):

Interrogator av Millen et al. Protokollen er modellert som en kommuniserende tilstandsmaskin hvor inntrengeren kan ødelegge, fange opp og forandre på alle meldinger. Gitt en slutt-tilstand, hvor inntrengeren har fått tilgang til en hemmelig

[1] Ina Jo er et formelt spesifiseringsspråk som er en utvidelse av første ordens predikat logikk (kalkulus).
[2] Jeg bruker forkortelsen ASM som kommer fra engelsk: Abstract State Machine.
[3] Abstract State Machine Language
[4] Isabelle er en generisk teorembeviser utviklet bl.a. for HOL.

beskjed, prøver Interrogator å konstruere angrep som fører frem til slutt-tilstanden. Det gjøres ved å søke igjennom et uendelig søkerom. Det er mulig fordi systemet kan begrense søkerommet sterkt bl.a. ved å finne uendelige løkker osv. Begrensningene må gjøres i samspill med brukeren av systemet. Prosessen er ikke helautomatisk, men den gjør at søkerommet kan begrenses nok til at det er mulig å gjøre et fullstendig søk.

NRL protocol Analyzer av Meadows et al. NRL bruker omtrent samme metode som Interrogator. Vi har en "uønsket" tilstand og prøver å finne alle stier frem til denne fra en starttilstand. Ulikt Interrogator vil NRL ikke bare prøve å finne stier til usikre tilstander, men også prøve å bevise at slike tilstander ikke kan nås. Når disse tilstandene blir eliminert gjør det at søkerommet blir passende for et uttømmende søk. NRL finner disse bevisene hovedsakelig ved hjelp av brukeren og er derfor mindre automatisert en Interrogator. NRL er likevel et godt verktøy da den også har funnet ukjente svakheter i autentiseringsprotokoller.

Longley and Rigby Longley og Rigby benytter seg av et regelbasert system hvor mål blir gjort om til submål som etterhvert danner et tre. I treet representerer hver node et datafelt. Barna til en node i treet representerer de data som er nødvendige for kunnskapen om de data som ligger i "foreldre"–noden. Slik er det mulig å lage et tre hvor rotnoden representerer data som en inntrenger er ute etter (f.eks. kryptografiske nøkler), og løvnodene representerer data som er nødvendige for å få kunnskapen i rotnoden. Slik kan en protokolldesigner bruke verktøyet for å legge inn data som en eventuell angriper har mulighet til å få tak i, og generere et tre på bakgrunn av det. Longley og Rigby sitt verktøy har bl.a. funnet hittil ukjente feil i hierarkiske nøkkelfordelingssystemer.

Problemet med ekspertsystemene er ofte at de er lite effektive, fordi de må utføre uttømmende søk i store søkerom. Noen ganger vil de gå i evige løkker og resultatene kan være ufullstendige. Fordelen med disse tilnærmingene er at hvis det er funnet en mulig angrepsvei, så er angrepet som gjør dette mulig direkte tilgjengelig, og en kan forbedre protokollen. Det er f.eks. ikke mulig med de modale logikkene vi skal se på i neste avsnitt.

6.4 Modallogikker

En annen måte å analysere sikkerhets og autentiseringsprotokoller på har vært å bruke modallogikker utviklet for resonnering om kunnskap og viten ("knowledge and belief") (Fagin, Halpern, Moses, og Vardi 1995). Slike modallogikker har vært brukt mellom annet til å analysere kunnskap og viten i distribuerte systemer. Logikkene består av et språk for å beskrive de forskjellige utsagnene vi har om protokollen, hvem av partene vet hva, osv. Det finnes også slutningsregler, som brukes for å utlede nye utsagn fra tidligere utlede utsagn.

En av de mest kjente og betydningsfulle logikkene i forbindelse med autentiseringsprotokoller har vært så kalt BAN–logikk (Burrows, Abadi, og Needham 1990). Oppkalt etter

de tre hovedmennene bak den; Burrows, Abadi og Needham. BAN–logikk kan spesifisere protokoller ved hjelp av utsagn om beskjedene som er sendt og mottat i en gjennomføring av protokollen. Slutningsregler brukes for å se hvordan en ved hjelp av noen initielle oppfatninger kan slutte seg til nye oppfatninger. Dersom mengden av oppfatninger er adekvat i forhold til en forhåndsbestemt norm, sier vi at protokollen er korrekt. Dersom det ikke stemmer, har vi muligheter for å finne feil i protokollen. Verifisering utføres for hånd. BAN–logikk er godt egnet for å studere autentiseringsprotokoller fordi den benytter seg av s.k. idealiseringer som fjerner en del distinksjoner som ellers kan skape problemer. Den prøver ikke å modellere det å stole på, distinksjoner mellom å se noe og å forstå noe, det å forandre hva en tror, den prøver heller ikke å modellere kunnskap. Det gjør logikken enkel og rett på sak. Disse distinksjonene må derfor behandles når en går fra en uformell spesifikasjon til en BAN spesifikasjon. Dette blir imidlertid også pekt på som en av svakhetene ved tilnærmingen. Arbeidet med BAN–logikk har vist at den er veldig effektiv dersom en tar hensyn til begrensningene og gjør klart hvilke forutsetninger som gjelder for de ulike protokollene. På grunn av kritikk som er rettet mot BAN–logikken, hovedsakelig fordi den ikke behandler kunnskap, og dermed ikke hemmeligholdelse, er det kommet flere utvidelser av logikken. En av disse er å utvide logikken til å ta for seg flere momenter, det finner vi bl.a. i GNY–logikk, utviklet av Gong, Needham og Yahalom. Utvidelsene av BAN–logikken har gjort GNY relativt kompleks, noe som kan skape problemer med bruk av den. Syverson og van Oorschot har sett på flere slike logikker for å prøve å forene de, og gi en felles semantikk. Slik kan logikkene anvendes uten for store problemer, samtidig som de er utrykksfulle nok.

I tillegg til BAN–logikk og dens utvidelser, har vi en del andre modallogikker som også har vært brukt til å se nærmere på autentiseringsprotokoller. Vi har Biebers CKT5, Syversons KPL, Mosers logikk m.fl.

6.5 Omskrivningslogiske og algebraiske systemer

En fjerde gruppe formelle metoder finner vi blant de som bruker ulike algebraiske systemer og omskrivningslogikk for å resonnere om kunnskap. Et av de første arbeidene utført innenfor dette området er Dolev og Yaos artikkel som beskriver protokoller som et algebraisk system (Dolev og Yao 1983). I Dolev og Yaos modell ser vi på et nettverk som er under fullstendig kontroll av en inntrenger. Inntrengeren kan samle inn beskjeder, ødelegge beskjeder og sende nye beskjeder. Inntrengeren kan også utføre operasjoner slik som andre legitime brukere har mulighet til. På bakgrunn av disse antagelsene ser Dolev og Yao på alle meldinger som blir mottatt av en legitim bruker av nettverket som om de kommer fra inntrengeren. Alle meldinger som en legitim bruker sender blir sett på som om de er sendt til inntrengeren. Systemet kan derfor sees på som å være en maskin for inntrengeren til å generere termer (ord). Ordene følger regler for omskrivning , og vi har et omskrivningssystem som kan brukes av inntrengeren. Hvis målet til inntrengeren er å komme fram til et ord som skal være hemmelig, reduseres problemet med å vise at en protokoll er sikker, til

problemet om det er mulig å bevise at et ord ikke kan genereres i omskrivningssystemet. Dolev og Yao modellen er ganske begrenset og omfatter bare noen typer av protokoller som de kaller henholdsvis "Cascade protocols" og "Name-Stamp protocols". Det er kun mulig å finne feil mht. hemmelighold, og partene kan ikke "huske" tilstandsinformasjon fra en tilstand til en annen.

Det er gjort arbeid for å utvide Dolev og Yaos system slik at det også kan brukes til å studere andre protokoller. Det er blant annet gjort av Merrit, som generaliserer teknikkene til å se på flere typer protokoller, og som også behandler andre spørsmål enn kun hemmelighold (DeMillo et al. 1982).

Nyere arbeid har sett nærmere på bruken av π–kalkulus. Abadi og Gordon har brukt π–kalkulus for å se på protokoller på et abstrakt nivå. Det blir brukt til å modellere visse egenskaper til protokollene, videre har de utvidet π–kalkulusen til noe de kaller Sπ–kalkulus. Sπ–kalkulus brukes til å analyserer protokollene på et mindre abstrakt nivå (Buttyan 1999). Vi har også et system for omskrivningslogikk kalt *Maude* som er brukt mye til protokoll-spesifikasjon, se Denker, Meseguer, og Talcott (1998).

6.6 Formelle metoder og abstraksjonsnivå

De ulike formelle metodene har alle sine fordeler og ulemper. De behandler protokollene på ulike abstraksjonsnivåer og med ulike forutsetninger. De fleste formelle metoder har endel forutsetninger som må tilfredstilles før de kan anvendes. Dette gjelder for eksempel kryptering. Når vi bruker et språk for å beskrive en protokoll, ser vi ofte på de kryptografiske funksjonene som "svarte bokser" hvor vi forventer at alt fungerer som det skal, og at vi ikke må bekymre oss for kryptoanalyse av disse algoritmene. Det er dessverre ikke alltid like enkelt å tilfredstille forutsetningene, noe som vi kan se i kritikken av BAN–logikk. I BAN–logikk går diskusjonen på hva som bør sees på som forutsetninger/idealiseringer og hva som bør behandles av logikken. Problemet med de idealiseringene som gjøres er at noen egenskaper kanskje ikke kan behandles på en god måte. Spørsmålene som stilles er blant annet:

1. Er det mulig å behandle en bestemt egenskap, og fortsatt kunne avgjøre ved hjelp av logikken om protokollen er sikker? Eller vil det å legge til denne egenskapen gjøre logikken for kompleks og lite egnet?

2. Hvor viktig er det å modellere denne egenskapen? Påvirker den sikkerheten til protokollen? Kan denne egenskapen behandles av andre formelle metoder?

3. Er det en naturlig abstraksjonsbarriere mellom de egenskapene som behandles og de som ikke behandles? Dersom to egenskaper er nært knyttet til hverandre, er det kanskje naturlig å behandle begge?

De forskjellige systemene vil finne ulike steder å sette grenser for hva som skal være med, og hva som ikke skal være med. Noen ganger er det mest nyttig å modellere en protokoll

med hjelp av BAN–logikk, som ser på intensjoner. Andre ganger er det mer nyttig med en mer detaljert analyse slik som f.eks. ved hjelp av *Interrogator* eller *NRL Protocol Analyzer*.

Det kan være enkelt å konkludere med at det er de mest detaljerte modellene som er mest nyttige. Det er ikke nødvendigvis sant da en mer abstrakt modell kan være et bedre startpunkt i design av en protokoll, enn det å gå rett ned i detaljene. Med en oversiktlig modell i et høynivåspråk er det mulig å avdekke en del fundamentale feil, før en har begynt på implementeringen. Slik er det mulig å utsette valget av en del implementasjonsmessige valg til en har en godt testet modell. Det er derfor nyttig i arbeidet med en ny protokoll å benytte ulike verktøy i ulike deler av designprosessen. Designerern starter med et høynivåspråk/logikk og fortsetter med andre metoder som ser på detaljer i implementasjonsøyemed etterhvert som arbeidet skrider frem. En slik tilnærming gjør oss i stand til å avdekke feil så tidlig som mulig, med minst mulig feil.

6.7 Sammendrag

Vi har sett på noen av de ulike formelle metodene som er i bruk ved design og verifisering av autentiseringsprotokoller. De behandler protokoller ulikt og det er også forskjeller i hva de kan finne ut. For å behandle alle sider av en protokoll fra design til verifikasjon og fra krypteringsalgortime til nøkler er det nødvendig å benytte flere av metodene. Ulike metoder passer til ulike bruksområder.

Kapittel 7

Needham-Schroeder

I sin banebrytende artikkel "Using Encryption for Authentication in Large Networks of Computers" presenterer Roger M. Needham og Michael D. Schroeder eksempler på protokoller for autentisering, kryptering av meldinger, digitale signaturer og dokumentintegritet (Needham og Schroeder 1978a). Fra denne artikkelen skal vi se nærmere på en autentiseringsprotokoll basert på asymmetrisk kryptering se kap. 3 side 17. Protokollen har fått navnet Needham-Schroeder protokollen selv om den er en av flere protokoller beskrevet i artikkelen. Protokollen brukes for at to parter som vil kommunisere med hverandre på et distribuert nettverk, skal kunne forsikre seg om hverandres identitet. Den gir en gjensidig autentisering av de kommuniserende partene. Protokollen benytter seg av offentlige nøkler og en pålitelig tredjepart. Flere varianter av denne protokollen er beskrevet i litteraturen, her skal vi se på to beskrivelser av noe som essensielt er samme protokoll.

Hver part har en s.k. "offentlig nøkkel" (en allment kjent nøkkel), K_a for part A, som en hvilken som helst annen part kan hente fra en nøkkel-server. Part A har også en "privat nøkkel" K_a^{-1}, som er den inverse av K_a. Det er veldig viktig at den private nøkkelen blir holdt hemmelig jf. avsnittet om kryptering ved hjelp av offentlige nøkler side 17 kap. 3. Vi skriver $\{M\}_k$ for meldinger M som er kryptert med nøkkelen k. En hvilken som helst agent kan kryptere en melding M med A sin "offentlige" nøkkel noe som gir $\{M\}_{Ka}$, men bare A kan dekryptere denne meldingen, og det er dette som sikrer hemmeligholdelse. A kan signere en melding M ved å kryptere den med sin private nøkkel $\rightarrow \{M\}_{Ka^{-1}}$; en hvilken som helst agent som har A sin "offentlige nøkkel" kan dekryptere beskjeden. Krypteringen med A sin private nøkkel skal forsikre andre agenter om at beskjeden virkelig kom fra A (elektronisk signatur).
Protokollen bruker også "Nonces": tilfeldige tall generert for å brukes kun en gang dvs. i en anvendelse (one run) av protokollen.
N_a – nonce generert av A.
N_b – nonce generert av B.

1. A $\rightarrow$ S : A,B
2. S $\rightarrow$ A : $\{K_b, B\}_{Ks^{-1}}$
3. A $\rightarrow$ B : $\{N_a, A\}_{Kb}$
4. B $\rightarrow$ S : B,A
5. S $\rightarrow$ B : $\{K_a, A\}_{Ks}^{-1}$
6. B $\rightarrow$ A : $\{N_a, N_b\}_{Ka}$
7. A $\rightarrow$ B : $\{N_b\}_{Kb}$

Figur 7.1.1: Needham-Schroeder protokollen asymmetrisk versjon

7.1 Needham-Schroeder protokollen i 7 trinn

Denne beskrivelsen er den som er brukt i original artikkelen til Needham og Schroeder (Needham og Schroeder 1978a), og som vi også finner hos Lowe (Lowe 1995).

Her er A den som starter sesjonen og ønsker å starte en "samtale" med B, ved hjelp av nøkkelserveren S. I steg 1, sender A en beskjed til serveren, kryptert med en hemmelig nøkkel delt av A og serveren S, hvor den ber om B sin offentlige nøkkel. S svarer i beskjed 2 ved å returnere nøkkelen K_b, sammen med B sin identitet (for å unngå angrep basert på å omdirigere nøkkelleveranser), kryptert med S sin private nøkkel (for å forsikre A om at beskjeden kommer fra S). A forsøker så å opprette en forbindelse med B ved å velge en nonce N_a, og sende den sammen med sin identitet til B (beskjed 3), kryptert med B sin offentlige nøkkel. Når B mottar beskjeden, dekrypterer B den for å finne nonce N_a. B sender en forespørsel om A sin offentlige nøkkel (beskjed 4) og mottar den i (beskjed 5). Den returnerer så nonce N_a, sammen med en ny nonce N_b, til A, kryptert med A sin offentlige nøkkel (beskjed 6). Når A mottar denne beskjeden er A sikker på at han kommuniserer med B, siden bare B skal være i stand til å dekryptere beskjed 3 for å få tak i N_a. A returnerer så nonce N_b til B, kryptert med B sin offentlige nøkkel. Når B mottar denne beskjeden skal han være sikker på at han kommuniserer med A, siden kun A skal være i stand til å dekryptere melding 6 for å få tak i N_b.

Protokollen kan sees på som en sammensmeltning av to logisk disjunkte protokoller: beskjed 1, 2, 4 og 5 sørger for distribusjonen av offentlige nøkler, mens beskjed: 3, 6 og 7 er relatert til autentiseringen mellom A og B. Denning og Sacco (1981) har poengtert at protokollen gir ingen garanti for at de kjente nøklene er nåværende og ikke "gjentakelser" av gamle, muligens kompromitterte nøkler. Det problemet kan overkommes på ulike måter, f.eks. ved å inkludere tidsstempel (timestamp) i nøkkelleveringen.

7.2 Angrep og feil - Needham-Schroeder

Needham-Schroeder protokollen er på mange måter en todelt protokoll. Beskjedene 1, 2, 4 og 5 tar for seg tilegnelse av offentlige nøkler. Beskjedene 3, 6 og 7 utfører selve autentiseringen. Vi forutsetter at hver agent til å begynne med har hverandre sine offentlige nøkler, og ser på de følgende meldingene:

3. A $\rightarrow$ B : $\{N_a, A\}_{Kb}$
6. B $\rightarrow$ A : $\{N_a, N_b\}_{Ka}$
7. A $\rightarrow$ B : $\{N_b\}_{Kb}$

Figur 7.2.1: Meldinger som tar for seg autentisering.

Vi vil se på hvordan en inntrenger kan interagere med protokollen (Lowe 1995). Inntrengeren I er en bruker i datanettverket og kan kommunisere på vanlig måte med andre agenter i datanettverket og andre kan sette opp kommunikasjon med I. Faktisk starter angrepet med at agent A prøver å sette opp en sesjon med I. Vi antar at inntrengeren kan fange inn meldinger på nettverket, og sende (introdusere) nye meldinger. Vi må også gjøre oss noen antakelser om hvilken type meldinger inntrengeren kan lage. Vi antar at inntrengeren ikke kan gjette verdien på nonces som er sendt i krypterte meldinger, dersom ikke disse beskjedene er kryptert med I sin nøkkel. Så inntrengeren kan kun lage nye meldinger med nonces som den har laget selv, eller som den tidligere har sett og forstått. Inntrengeren kan også repetere hele krypterte meldinger selv om den ikke forstår innholdet. Angrepet på protokollen gir en Inntrenger I, muligheten til å utgi seg for å være en agent A og sette opp en falsk sesjon med B. Angrepet involverer to "runder" (runs) av protokollen. I første runde etablerer A en gyldig sesjon med I, i den 2. runden vil I utgi seg for å være A og slik kunne etablere en falsk sesjon med B. I eksempelet under vil 1.3 stå for beskjed nummer 3 i 1. runde; vi skriver I(A) for å representere at Inntrengeren I gir seg ut for å være A:

1.3. A $\rightarrow$ I : $\{N_a, A\}_{K_i}$

2.3. I(A) $\rightarrow$ B : $\{N_a, A\}_{K_b}$

2.6. B $\rightarrow$ I(A) : $\{N_a, N_b\}_{K_a}$

1.6. I $\rightarrow$ A : $\{N_a, N_b\}_{K_a}$

1.7. A $\rightarrow$ I : $\{N_b\}_{K_b}$

2.7. I(A) $\rightarrow$ B : $\{N_b\}_{K_b}$

I steg 1.3, starter A med å etablere en normal sesjon med I, ved å sende nonce N_a. I steg 2.3 gir inntrengeren seg ut for å være A for å etablere en falsk sesjon med B, dette gjøres

ved å sende nonce N_a som den fikk i forrige melding. B svarer i melding 2.6 ved å velge en ny nonce N_b, og prøver å returnere den sammen med N_a, til A. Inntrengeren snapper opp denne meldingen, men kan ikke dekryptere den fordi den er kryptert med A sin nøkkel. Inntrengeren prøver derfor å bruke A som et orakel ved å videresende meldingen til A i beskjed 1.6; legg merke til at meldingen har den form som A forventer i første runde av protokollen. A dekrypterer meldingen for å tak i N_b, og returnerer denne til I i melding 1.7. I kan så dekryptere denne meldingen for å få tak i N_b, som den returnerer til B i beskjed 2.7, og avslutter dermed 2. runde av protokollen. B tror dermed han har satt opp en korrekt sesjon med A.

Konsekvenser av angrepet Hva er konsekvensene av dette angrepet? Det er foreslått at fordi noncene er s.k. *delte hemmeligheter*, så kan de inkluderes i etterfølgende meldinger som autentisering se (Needham og Schroeder 1978a)og (Burrows, Abadi, og Needham 1990). Det vi ser er at etter det ovenfornevnte angrepet så vet inntrengeren noncene så han kan fortsette å utgi seg for å være A og sende meldinger til B under sesjonen. Eksempelvis kunne inntrengeren inkludere noncene i en melding og foreslå en sesjonsnøkkel, og B vil tro at denne meldingen kom fra A. Dersom B var en bank kunne I sende meldinger av typen: $I(\text{A}) \rightarrow \text{B} : \{N_a, N_b, \text{“Overfør £1000 fra min konto til Is”}\}_{K_b}$

Erfaringer Vi har fått presentert et angrep på den velkjente Needham-Schroeder public-key autentiseringsprotokollen. Angrepet gjør en inntrenger i stand til å utgi seg for å være en bestemt part i en kommunikasjonssituasjon mellom to parter. Det er relativt enkelt å forandre protokollen slik at angrepet kan unngås. Vi kan inkludere identiteten til den som svarer i beskjed nummer 6 i protokollen:

6. $\text{B} \rightarrow \text{A} : \{B, N_a, N_b\}_{K_a}$

da vil steg 2.6 bli:

2.6. $\text{B} \rightarrow \text{I(A)} : \{B, N_a, N_b\}_{K_a}$

og inntrengeren kan ikke bruke denne beskjeden i 1.6, fordi A forventer en beskjed med I sin identitet.

7.3 Angrep på nøkkeldistribusjon

Vi har også andre typer av angrep som er rettet mot Needham-Schroeder protokollen. Som vi har sett ovenfor er dette angrepet rettet hovedsakelig mot selve autentiseringsbiten av protokollen, vi har ikke sett nærmere på nøkkeldistribusjonen. Et angrep som retter seg mot denne delen av protokollen er beskrevet av Denning og Sacco (Denning og Sacco 1981). En ting som kan være verdt å merke seg er at Needham-Schroeder finnes i flere ulike

varianter, noen hvor det brukes symmetrisk kryptering og noen hvor det brukes asymmetrisk kryptering. Den versjonen vi skal se på nå benytter seg av symmetrisk kryptering og forutsetter derfor at nøklene som brukes holdes hemmelig og at både server S og en av partene A, B har en slik felles hemmelig nøkkel som de har gjort en trygg utveksling av på et tidligere tidspunkt. Det som er kjernen i dette angrepet er ikke "replay" (gjensendelser) av gamle meldinger, men eventuelle kompromitterte kommunikasjonsnøkler.

7.3.1 Denning og Saccos beskrivelse av Needham-Schroeder protokollen

For at A og B skal kunne autentisere hverandre, er det nødvendig med en felles hemmelig sesjonsnøkkel. Protokollen starter med at A henvender seg til en nøkkelserver S, for å få tak i sesjonsnøkkelen som skal brukes i autentiseringen mellom A og B. En forutsetning er som sagt at A og S har en felles hemmelig nøkkel (K_a), som kommunikasjonen mellom de kan krypteres med.

1. A $\rightarrow$ S : A, B, I_A
2. S $\rightarrow$ A : $\{I_A, B, CK, Y\}_{K_a}$

I melding 1 sender A et identitetselement I_A som brukes bare en gang (e.g. en nonce), i tillegg til A og B sine identiteter. Serveren S svarer med å sende tilbake identitetselementet, B sin identitet, en sesjonsnøkkel CK og Y = $\{CK, A\}_{K_b}$. Dette for å forsikre A om at det er serveren som svarer, og at det ikke er en gjentagelse av en tidligere sendt melding. A mottar og sender videre Y som inneholder sesjonsnøkkelen kryptert med B sin hemmelige nøkkel:

3. A $\rightarrow$ B : Y

Etter steg (3) vet vi at nøkkelen CK trygt kan brukes. Det som gjenstår nå er å forsikre B om at beskjed Y og de etterfølgende beskjedene som tilsynelatende kommer fra A ikke er gjentakelser av tidligere meldinger. Vi vil ha en gjensidig autentisering som innbefatter begge parter. For å beskytte mot gjensendelsesangrep("replays") benytter en seg av et s.k. håndtrykk mellom A og B som følger:

4. B $\rightarrow$ A : $\{I_B\}_{CK}$
5. A $\rightarrow$ B : $\{f(I_B)\}_{CK}$

hvor I_B er en identifikator som er valgt av B. A viser at han ønsker å bruke CK, ved å returnere en verdi $f(I_b)$, hvor f er en funksjon de tidligere er blitt enig om. f kan være så enkel som $f(I) = I - 1$. Sekvensen fra (1.) til (5.) etablerer en autentisert og sikker kommunikasjonskanal mellom A og B, så lenge tidligere kommunikasjonsnøkler og private nøkler ikke har blitt avslørt.

7.3.2 Kompromitterte kommunikasjonsnøkler

Dersom det brukes gode krypteringsalgoritmer og tilfeldige nøkler er det usannsynlig at kommunikasjonsnøklene kan avsløres ved hjelp av kryptoanalyse. Vi skal se nærmere på problemer med kommunikasjonsnøklene og eventuelle avsløringer av disse som følge av designfeil i system/protokoll, i.e. er det mulig for en inntrenger å f.eks. bryte seg inn i AS og stjele en nøkkel.

Eksempel på et angrep. Vi tenker oss at en tredjepart har avlyttet og lagret alle meldingene mellom A og B i stegene (3)–(5). Vi tenker oss også at C har fått tak i kommunikasjonsnøkkelen CK. C kan da lure B til å bruke CK som følger: Først gjentar C beskjed Y til B:

2.3 C $\rightarrow$ B : $\{CK, A\}_{K_b}$

Siden B tror A ønsker å sette opp en kommunikasjon vil B sette i gang med håndtrykksprosedyren:

2.4 B $\rightarrow$ C(A) : $\{I'_b\}_{K_b}$

C snapper opp meldingen, dekrypterer den og sender tilbake noe som tilsynelatende er A sin respons:

2.4 C(A) $\rightarrow$ B : $\{f(I_B)\}_{CK}$

Etter dette kan C sende falske meldinger til B som tilsynelatende kommer fra A, C kan snappe opp og dekryptere svarene B sender til A. Det er viktig å merke seg at A kan sette opp en ny og sikker autentisering/kommunikasjonssituasjon mot B dersom ikke C blokkerer dette. Det er heller ingenting i veien for at B kan autentisere seg og kommunisere sikkert med andre parter. Det som ikke er mulig for C, er å gjenta As respons til B i håndtrykket uten å vite om hvilken funksjon f A og B er blitt enige om å bruke. Problemet med å holde håndtrykksfunksjonen hemmelig er prinsipielt like vanskelig som det er å holde kommunikasjonsnøklene hemmelig. Enten må A og B møtes fysisk og bli enige om hva funksjonen f skal være, eller så må de få tak i f fra serveren S. Den første måten å gjøre det på kan være veldig vanskelig i praksis, men de kan da eventuelt utveksle kommunikasjonsnøklene direkte og kvitte seg med håndtrykksfunksjonen, dersom de benytter seg av serveren S, er problemet med å distribuere håndtrykksfunksjonen identisk med problemet i å distribuere nøklene. Problemet kan løses dersom de private nøklene er hemmelige og vi kan kvitte oss med håndtrykket ved hjelp av tidsstempler. Tidsstempler vil selv om de løser noe av problemet kunne skape problemer i det at de er avhengige av synkroniserte klokker, noe som kan være et problem dersom klokkene ikke er synkroniserte. Denning og Saccos løsning baserer seg på at klokkene må være riktige innenfor et visst intervall for å kunne sikre integriteten til tidsstemplene i protokollen.

Det som Denning og Sacco løser ved å innførte tidsstempler er problemet med gjentagelser av kompromitterte kommunikasjonsnøkler, og de kan også brukes for å slippe å gå igjennom en håndtrykksprosedyre (Denning og Sacco 1981).

7.4 Sammendrag

Vi har sett på hvordan Needham-Schroeder protokollen for autentisering fungerer, og vi har sett nærmere på angrep rettet mot to versjoner av protokollen. Den ene benytter seg av asymmetrisk kryptografi, den andre benytter seg av symmetrisk. Selv om Needham-Schroeder er en relativt liten protokoll dersom vi ser på antallet meldinger og innholdet i de, så er det interessant å se hvor mange feil og angrep de kan utsettes for. Dette kommer klart frem dersom vi ser på når feil og angrep mot protkollen har blitt publisert. Artikkelen til Needham og Schroeder kom i 1978, Denning og Sacco kom med sin kritikk i 1981, mens Lowes angrep mot den symmetriske versjonen kom først i 1995. Dette viser hvor vanskelig arbeidet med å formalisere og teste slike protokoller er. Vi trenger verktøy for både å vise korrekthet i forhold til feil så vel som angrep. Når vi har en spesifikasjon som er grundig testet, er det så en kritisk fase i overgangen fra spesifikasjon til implementasjon. Her er det nødvendig med gode verktøy som gjør denne overgangen så smidig og god som mulig.

Kapittel 8

ASM

Dette kapittelet inneholder en beskrivelse av abstrakte tilstandsmaskiner, fra nå av ASM. Først skal vi se nærmere på hvilke motivasjoner som ligger bak bruken av ASMer. Vi skal se på ASM sine formelle definisjoner med forklaringer, deretter ser vi på to eksempler på ASMer etterfulgt av en liten del om bruken av ASM i akademia og industri. Helt til slutt nevner vi hvilke verktøy som er utviklet for å støtte bruken av ASM noe vi kommer tilbake til i kap. 9.

8.1 Hvorfor bruke ASM

Abstrakte tilstandsmaskiner har et stort bruksområde, og vi skal komme tilbake til hvordan en slik abstrakt tilstandsmaskin ser ut. Vi begynner dette kapittelet med å se nærmere på hvilke motivasjoner som ligger bak bruken av ASMer. Formelle spesifikasjoner brukes ofte for å fange inn kravspesifikasjoner i et komplett og nøyaktig språk. ASM kan brukes for å gå fra naturlig språk med de begrensninger og rom for misforståelser det har, til en komplett og minimal beskrivelse med matematisk presisjon. Selve formaliseringen av et program, en protokoll eller maskinvare må løse tre problemer (Börger og Stärk 2003):

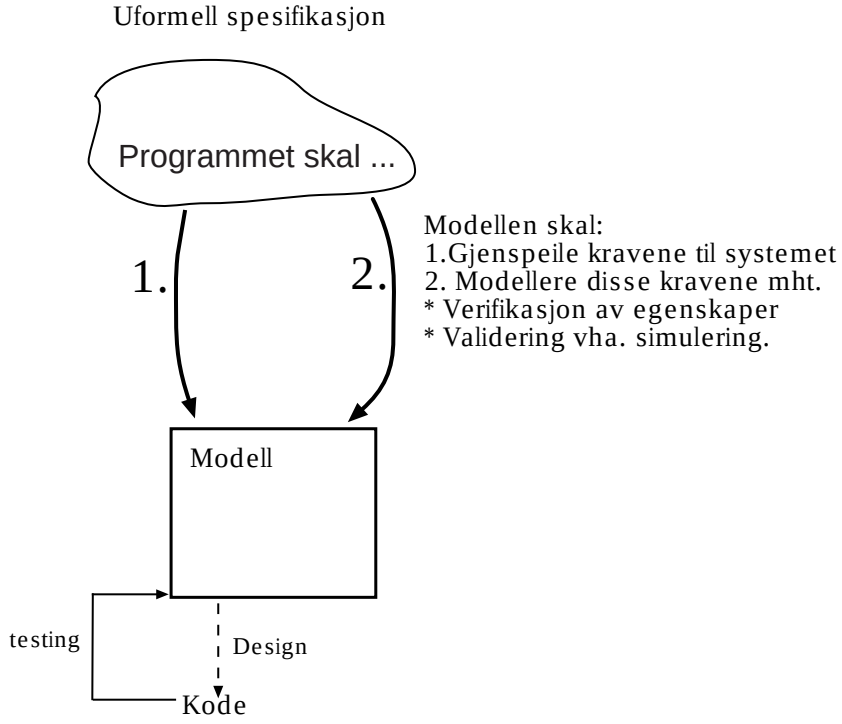
Språk og kommunikasjonsproblemet Vi må "oversette" mellom en uformell beskrivelse i naturlig språk eller domenespesifikt språk til en presis matematisk modell som kan benyttes i utviklingen av programmet, protokollen eller maskinvaren. Dette er et grunnlagsproblem og noe som ikke nødvendigvis kan løses. Språket som brukes for å beskrive modellen må være både rikt nok til å beskrive de nødvendige delene, samtidig som det må være presist nok slik at misforståelser unngås. Ulike problemer vil kreve ulike grader av abstraksjon. Vi ønsker at modellen skal på en mest mulig naturlig måte gi oss en generell data-modell, sammen med en funksjonsmodell (for å definere systemets dynamiske egenskaper), og et passende brukergrensesnitt mot omverdenen (brukere og andre systemer). Dette skal ikke gå på bekostning av lesbarhet, eller enkel tilegnelse av modellen. Det er derfor nødvendig å bestemme seg

for hva som er et riktig abstraksjonsnivå for en gitt modell. Vi ønsker å behandle det som er nødvendig og ikke noe mer.

Verifikasjons-metodeproblemet Vi må kunne kvalitetsikre overgangen fra en uformell beskrivelse til en presis beskrivelse. Det er derfor nødvendig med metoder og verktøy for å sikre at modellen er komplett og konsistent og at den reflekterer de originale intensjonene, og at disse har kommet med på en riktig måte. Modellen må derfor kunne leses og forstås av en domeneekspert for å sikre at modellen er så riktig som mulig. Vi ønsker også å tilby en domeneekspert hjelp i form av verktøy for å kunne formelt sjekke at en modell er komplett og internt konsistent. Dersom ikke modellen vi skal bygge systemet vårt på er komplett og internt konsistent, kan dette skape store problemer når vi går løs på selve implementasjonen av problemet vi skal løse. En metode som kan hjelpe oss i dette er å benytte seg av ASMer. ASMen kan benytte et passende abstraksjonsnivå for domenet vi skal løse problemet fra. Korrekthet kan bestemmes ved hjelp av inspeksjon av modellen og det er også mulig å vise kompletthet.

Valideringsproblemet Vi ønsker også å kunne kjøre simuleringer på modellen for å teste ulike scenarier. Det er ofte nødvendig å se hva som skjer i ulike tenkte situasjoner som oppstår ved kjøring av programmet. Ved hjelp av ASMer er det mulig å opprette et "orakel". Vi kan simulere kjøringer av valgte input for å se hva som vil skje i en implementasjon. Dette kan gjøres både ved statisk testing og dynamisk testing. Med statisk testing tenker vi på kodeinspeksjon og sammenligning med modellen. ASM støtter også dynamisk testing hvor resultater fra kjøring kan sammenlignes. En slik modell kan også brukes for å finne hva som er interessante eller kompliserte deler av systemet som skal undersøkes nærmere. Simuleringer av ASM er mulig fordi de kan kjøres både ved bruk av verktøy og "mentale" simuleringer. Dette gjør at ASM modellen kan fungere i rollen både som (1) en nøyaktig kravspesifikasjon og (2) som en test modell se del 8.1.1 side 57. Dette siste er noe som er foreslått innen Extreme Programming paradigmet [1].

[1] Se websiden www.extremeprogramming.org for mer informasjon.

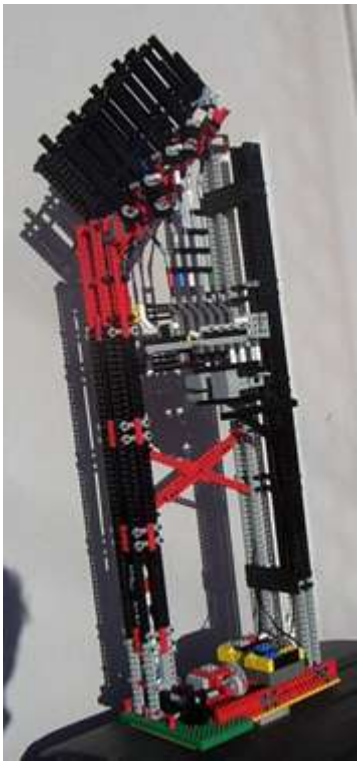


Figur 8.1.1: Bruk av ASM

8.2 Hvordan simulerer en ASM en algoritme?

ASMer gir operasjonell semantikk for algoritmer ved å bygge videre på Turings tese: Enhver beregning utført av en menneskelig beregner på en tape kan simuleres med en turingmaskin. Med simulering mener vi ikke bare at de riktige resultatene kan oppnås, men at turingmaskinen utfører de samme regnetrinn steg for steg. Problemet med simulering av algoritmer på en Turingmaskin er at et steg (en transisjon) i algoritmen tilsvarer mange steg (mange transisjoner) i Turingmaskinen. En Turingmaskin bruker mye tid på å kode/-dekode på grunn av regnemediet. En Turingmaskin benytter seg av en (tenkt) uendelig tape. På denne tapen blir et og et symbol lest eller skrevet. Hvert steg i en Turingmaskin består av: lese et symbol på tapen, skrive et symbol, gå et steg til venstre eller gå et steg til høyre. Turingmaskiner er teoretiske maskiner, men de er veldig kraftige, det kan vises at enhver beregning som er mulig å utføre kan utføres av en Turingmaskin. Selv om den benytter seg av noen veldig enkle operasjoner er den en universell beregningsmaskin. Dette gjør den veldig interessant utifra et teoretisk synspunkt. Dessverre er den tungvint å bruke og er derfor uegnet til å simulere algoritmer og beregninger bortsett fra de aller enkleste.

Selv om Turingmaskiner først og fremst er interessante rent teoretisk, finnes det mange eksempler på ulike implementasjoner slik som vi kan se i 8.2.1, hvor en Turingmaskin er implementert i Lego!



Figur 8.2.1: Turing maskin implementert i LEGO

Vi ser etter maskiner som kan simulere algoritmene på deres nivå. Vi vil ha et på forhånd oppgitt antall steg for hvert steg i algoritmen, ("lock-step") og vi vil ikke ha et fast abstraksjonsnivå. ASMer beskriver algoritmer på selvvalgte abstraksjonsnivåer, det er mulig å ha et hierarki av ulike abstraksjonsnivåer for den samme algoritmen. Slik kan en selv bestemme seg for det abstraksjonsnivå som passer best for problemet en jobber med (se 8.7). Videre vil vi se hvordan en ASM kan beskrives som tilstander og transisjoner mellom dem. Alt dette leder opp til Gurevich tese: Enhver sekvensiell beregning kan utføres ved en abstrakt tilstandsmaskin. Med utføres menes ikke bare at de riktige resultatene blir oppnådd, men også at beregningene svarer til hverandre trinn for trinn ("lock-step") (Jervell 2001).

8.3 Hvordan kan vi vise egenskaper ved algoritmer

Begynn med å beskrive algoritmen på et passende abstraksjonsnivå med en ASM A_0. Vis at formaliseringen er korrekt. Spesifiser så forbedrede (refinement) ASMer A_i som formaliserer ulike aspekter av algoritmen i mer detalj. Vis at alle A_{i+1} korrekt implementerer Ai.

8.4 En algoritme som ASM

Algoritmen starter i en starttilstand S_0 og går igjennom tilstandene S_1, S_2, osv. En tilstand er representert av en statisk algebra. Vi går fra S_i til S_{i+1} ved å gjennomføre et avgrenset antall steg i S_i. Disse stegene kan formuleres som transisjonsregler. En ASM består altså av tilstander og transisjoner mellom tilstander. Vi vil nå se nærmere på hvordan vi kan beskrive tilstander og transisjoner. Vi starter med tilstander. Det som følger her er en detaljert definisjon av noen av bestandelene til ASMer som også gir en innsikt i deres matematiske bakgrunn [2].

8.5 Tilstander hos en ASM - statiske algebraer

For å beskrive tilstander bruker ASMene spesielle algebraer kalt statiske algebraer. De bygger på Tarskis definisjon av første ordens strukturer. Strukturer uten relasjoner kalles algebraer i den matematiske disiplinen universal algebra. Vi starter med å se nærmere på slike statiske algebraer.

Definisjon av en statisk algebras signatur:

Definisjon 1 En *signatur* Σ er en endelig samling funksjonsnavn. Hvert funksjonsnavn f har en *aritet*, et ikke-negativt tall. Funksjoner med aritet null kalles *konstanter*. Funksjonsnavn kan være statiske eller dynamiske. En ASMs signatur forventes også å inneholde de statiske konstantene *true*, *false* og *undef*.

Signaturer blir også noen ganger kalt vokabularer. Ariteten til en funksjon angir hvor mange argumenter den tar.

Eksempel 8.5.1 *Signaturen Σ_{bool} til Boolsk algebra inneholder konstantene 0 og 1, en unær funksjon '-' og to binære funksjoner '+' og '*'.*

Definisjon av en statisk algebras tilstand:

Definisjon 2 En *tilstand* $\mathfrak{U}$ for signaturen Σ er en ikke-tom mengde X, *superuniverset* til $\mathfrak{U}$, sammen med *tolkninger* av funksjonsnavnene til Σ. Hvis f er en n-ær funksjon til Σ, så er dens tolkning $f^{\mathfrak{U}}$ en funksjon fra X^n til X. Dersom c er en konstant av Σ , så er dens tolkning $c^{\mathfrak{U}}$ et element av X. Superuniverset X av tilstanden $\mathfrak{U}$ er betegnet av $|\mathfrak{U}|$.

Superuniverset til en tilstand kalles også for basismengden av tilstanden. Elementene til en tilstand er elementene til superuniverset. Konstanten *undef* representerer et ikke-bestemt objekt, default verdien til *superuniverset*. *Superuniverset* kan deles opp i *subuniverser* som representeres ved unære relasjoer.

[2] Definisjonene i dette kapittelet er hentet og oversatt fra s. 63–86 i (Börger og Stärk 2003).

Eksempler på noen universer:

- Univers Ukedag Ukedag(mandag)=*true*, Ukedag(tirsdag)=*true*, ...
- Univers Bool Bool(true)=*true*, Bool(false)=*true*
- Univers Integer Integer(0)=*true*, Integer(1)=*true*,...

Formelt tolkes funksjonsnavn i tilstandene som totale funksjoner. Vi ser derimot på de som partielle og definerer *domenet* til en n-ær funksjon f i tilstanden $\mathfrak{U}$ til å være mengden av alle n-tupler $(a_1, ..., a_n) \in |\mathfrak{U}|^n$ slik at $f^{\mathfrak{U}}(a_1, ..., a_n) \neq undef^{\mathfrak{U}}$.

En *relasjon* er en funksjon som har alltid har en av verdiene *true*, *false* eller *undef*. Vi kan tenke oss en n-ær relasjon som R som mengden av alle n-tupler $(a_1, ..., a_n)$ slik at $R(a_1, ..., a_n) = true$.

Definisjon av en statisk algebras posisjon

Definisjon 3 En posisjon eller plassering av $\mathfrak{U}$ er et par $(f(a_1, ..., a_n))$. hvor f er en n-ær funksjon og $a_1, ..., a_n)$ er elementer fra $|\mathfrak{U}|$. Verdien $f^{\mathfrak{U}}(a_1, ..., a_n)$ kalles innholdet til posisjonen i $\mathfrak{U}$. Elementene til posisjonen er elementene i mengden $a_1, ..., a_n$.

En tilstand $\mathfrak{U}$ kan sees på som funksjonen som assosierer posisjonene til $\mathfrak{U}$ til dens innhold. Vi skriver $\mathfrak{U}(l)$ for innholdet av posisjonen l i $\mathfrak{U}$.

8.6 Transisjoner hos en ASM

Dersom dette er en tilstand, hvordan foregår da transisjonene mellom disse? Hvordan forandres disse strukturene?

Transisjons regler

- Transisjons-regler er ansvarlig for å gå fra en statisk algebra til en annen.
- Etterfølgende anvendelser av transisjons-regler definerer en slags eksekvering, hvor tilstandene er gitt som statiske algebraer [3].

[3] På bakgrunn av dette var det opprinnelige navnet på ASMer "evolving algebras" se feks. (Gurevich 1993)

8.6.1 Lokal funksjonsoppdatering

- Den eneste primitive transisjonsregelen er "lokal funksjonsoppdatering": $f(t_1, ..., t_r) := t_0$ hvor - f er en basisfunksjon med aritet r, og alle t_i er termer som evaluerer til et element a_i i superuniverset.
- En lokal funksjonsoppdatering - eller bare oppdatering - gir en ny verdi til funksjonen på spesifisert posisjon. Eksempel:

```
Tilstand $S_{i}$:
f 4,14,7,5
  2,12,6,1

f(7) := 3

Tilstand $S_{i+1}$:
f 4,14,7,5
  2,12,3,1
```

Eksempelet viser hvordan vi kan gi verdien 3 til funksjonen i posisjon 7.

Oppdatering og oppdateringsmengde, formell notasjon:

Definisjon 4 En *oppdatering* for $\mathfrak{U}$ er et par (l, v) hvor l er posisjonen i $\mathfrak{U}$ og v er et element av $|\mathfrak{U}|$. Oppdateringen er *triviell* dersom v er innholdet av l i $\mathfrak{U}$. En oppdateringsmengde er en mengde med oppdateringer.

Med andre ord betyr en oppdatering at innholdet til posisjonen l i $\mathfrak{U}$ forandres til verdien v. En oppdatering spesifiserer hvordan funksjonstabellen til en dynamisk funksjon oppdateres på de korresponderende posisjonene. Siden parallelismen kan gjøre flere oppdateringer av en funksjons argumenter flere ganger i en transisjon, må vi kreve at disse oppdateringene er konsistente. To oppdateringer kan *kollidere*, hvis de referer til samme posisjon men er forskjellige.

Definisjon 5 En oppdateringsmengde U er konsistent hvis den ikke har noen kolliderende oppdateringer, i.e, hvis for en hvilken som helst posisjon l og alle elementer v, w, er det sant at hvis $(l, v) \in U$ og $(l, w) \in U$, så er $v = w$.

8.6.2 Voktede oppdateringer (Guarded updates)

- En voktet oppdatering er en transisjonsregel på følgende form: *if b then u endif* hvor - b (vokteren) er en hvilken som helst term og - u er hvilken som helst oppdatering.
- Hvis b evaluerer til sann på den gitte statiske algebra så gjør u; ellers gjør ingenting.

8.6.3 Eksekvering av oppdateringer

Alle oppdateringer som utføres for en gitt statisk algebra S utføres *i parallell*. Oppdatereren (the Updator) utfører oppdateringene. Problem: han glemmer alltid oppdateringene som han allerede har utført, husker bare tilstand S_i.

- De følgende oppdateringer bytter verdiene til a og b:

```
a:=b
b:=a
```

- Inkonsistente oppdateringer gjør at beregningen stopper (kræsjer):

```
a:=true
a:=false
```

I matematikk er tilstander, slik som Boolsk algebra, statisk; de forandrer seg ikke. Innen databehandling er tilstander dynamiske; de forandres over tid. Dette skjer ved at de oppdateres underveis i beregningen. Å oppdatere abstrakte tilstander vil si å forandre tolkningen av (noen av) funksjonene til den underliggende signaturen. Måten en ASM oppdaterer tilstander er beskrevet i transisjonsreglene som definerer syntaksen til ASM programmet.

Definisjon av ASM formell notasjon:

Definisjon 6 En *abstrakt tilstandsmaskin M* består av en signatur Σ, en mengde initial tilstander for Σ, en mengde med regeldeklarasjoner og en spesiell regel med aritet 0 som vi kaller *hovedregelen* for maskinen. I en gitt tilstand, vil en transisjonsregel til en ASM gi hver variabeltildeling en oppdateringsmengde.

Statisk algebra: "Reserve"

- "Reserve"-universet *Reserve* fungerer som en uendelig ressurs for å lage nye elementer.
- Importerte elementer tas fra *Reserve* og legges til et hvilket som helst annet Univers. Notasjon: import x ... endimport

8.7 Eksempel 1: Største felles faktor

La oss se på et eksempel på hvordan en kommer fram til en ASM for et gitt problem, eksemplet har vi hentet fra (Huggins og Wallace 2002). I matematikkens verden har vi ofte bruk for å regne ut største felles faktor (greatest common divisor på engelsk). Hvilke univers trenger vi for å beskrive en algoritme for å finne sff (største felles faktor)? Vi

trenger sannsynligvis mengden med alle positive heltall. La oss kalle det universet *PosHel*. Noe som kan skape problemer her er kanskje at universet PosHel er uendelig. Dette er noe vi må ta hensyn til ved en eventuell implementasjon, men i en ASM modell trenger vi ikke ta hensyn til dette. ASM modellen må også inneholde selve utregningen av sff. Vi kan tenke oss at vi har en funksjon *sff* som tar to *PosHel* **a** og **b** og returnerer en *PosHel*. Dette er selvsagt en forenkling da vi tar for gitt at vi har en algoritme *sff*. Det er likevel en start for videre arbeid. Dette er også et godt eksempel på et høyt nivå av abstraksjon ved beskrivelse av en algoritme. Vi kan se på det som en første spesifikasjon av algoritmen. La oss gå videre og se på hvilken metode vi skal bruke for å regne ut sff. Vi kan feks. bruke Euklids algoritme [4]. Hvilke operasjoner trenger vi for å benytte Euklids algoritme? Vi trenger ihvertfall modulus operasjonen. Vi bruker infiks formen slik: **a** *mod* **b**. En ting som er verdt å legge merke til er at operasjonen modulus kan returnere 0. Vi trenger derfor å utvide ASM modellen vår med universet til de naturlige tallene, vi kaller universet *Nat*. Vi har altså funksjonen *mod* som tar inn to *PosHel* og returnerer et *Nat*. Dette kan vi skrive som: $PosHel \times PosHel \rightarrow Nat$ i matematisk funksjonsnotasjon. Vi trenger også en funksjon som gir oss likhet. Vi kan definere den som $Nat \times Nat \rightarrow Boolean$. En funksjon som gitt to naturlige tall gir oss **true** eller **false**. Hvor *Boolean* er universet med boolske variable. Det er også vanlig å bruke a=b som en forkortelse for funksjonen *Equal*(a,b). Hva mer trenger vi for å lage ASMen for algoritmen? Vi trenger noen steder å "oppbevare" midlertidige resultater. Vi kan feks. definere *A*, *B* og *Output* som funksjoner uten aritet for å oppbevare *PosHel* under beregningen. Nå har vi definert universer og funksjoner og kan gå over til å se nærmere på selve programmet. Euklids algoritme benytter seg av den matematiske identiteten *sff(a,b) = sff(b, a mod b)*, gitt at $a \bmod b \neq 0$. Hvis $a \bmod b = 0$, så vil *b* dele *a* og *sff (a,b) = b*. Når vi har dette på plass la oss se på et første program for Euklids algoritme:

```
if Output = undef then
  if(A mod B = 0) then Output := B
  else
      A:=B
      B:=A mod B
  endif
endif
```

Figur 8.7.1: ASM kode sff 1

Merk at **:=** betegner en tilordning. Her legger vi merke til paralleliteten i linjene hvor **A:=B** og **B:=A mod B**. Dette programmet gir oss en ASM for et gitt abstraksjonsnivå.

[4] Euklids algoritme kan defineres som:
Største felles faktor for tallene a og b større enn 0:
Så lenge a>0:
hvis a<b : bytt a og b
a = a-b

Hvilke andre nivåer av abstraksjon kan vi tenke oss? På det høyeste nivå kan vi tenke oss programmet som:

```
if Output = undef then Output := sff(A,B) endif
```

eller til og med:

```
Output := sff(A, B)
```

Forskjellen på disse to programmene er hvordan oppdateringen av Output foregår. I det første programmet blir Output oppdatert kun en gang med det riktige resultatet. I den andre oppdateres Output flere ganger med riktig resultat. Dersom vi ser på et annet mer ekstremt tilfelle andre veien, så kan vi tenke oss at vi ikke har tilgang til noen modulus operasjon. Vi må derfor erstatte den delen av ASMen som vi så på i figur 8.7.1 som benytter seg av modulus operasjonen med en mer detaljert ASM. Vi må regne ut verdien $A \ mod \ B$ ved å starte med A og kontinuerlig trekke fra B til vi får noe midre enn B. Vi tenger også et sted å lagre det midlertidige resultatet. La oss definere det som $AmodB$. Da får vi noe slikt som:

```
if CurrentMode = start then
  AmodB := A
  CurrentMode := calculateAmodB
elseif CurrentMode = calculateAmodB then
  if AmodB < B then CurrentMode := doEuklid
  else AmodB := AmodB - B
  endif
elseif CurrentMode = do Euclid then
  if AmodB = 0 then
     Output := B
     CurrentMode := done
  else
     A := B
     B := AmodB
     CurrentMode := start
  endif
endif
```

Figur 8.7.2: ASM kode sff 2

Her kan vi se at det er sneket inn noen ekstra funksjoner; vi har substraksjon, mindre-enn relasjonen ($<$) og en konstant CurrentMode. Denne ASMen vil gå i mellom de ulike tilstandene flere ganger før den stopper i slutttilstanden som er angitt med *CurrentMode := done*. Vi kan se på denne siste ASMen som en kombinasjon av to midre ASMer. Den

ene regner ut *A mod B* som vi kan se i de to første delene av if regelen, den andre utfører Euklidsalgortime og gir riktig verdi til *AmodB*. For detlajer omkring ASMene for Euklids algoritme og hvordan vi kom fram til disse se dialogen mellom Questor og Author i (Huggins og Wallace 2002)

8.8 Eksempel 2: En stakk-automat

Vi ser på en stakk-automat for å gjøre beregninger på uttrykk gitt i "reverse Polish notation"(RPN), denne skal vi modellere.

```
Et uttrykk
(1+23)*(45+6)
skrevet i RPN vil se ut som
1 23 + 45 6 + *
```

Uformell beskrivelse av en stakk-automat. Basis mengdene (kodet som universer) er "Data" og "Oper". Data er mengden med heltall, og Oper er mengden {+,*}. RPN uttrykket er gitt i form av en liste hvor hver innførsel er et datum eller en operasjon. Maskinen leser en innførsel fra lista igangen fra input fila. Dersom innførselen er et datum, blir det "pushet"på stakken. Dersom innførselen er en operasjon, vil maskinen "poppe"to ting fra stakken, anvende operatoren og "pushe" resultatet tilbake på stakken. Stakken er tom til å begynne med.

```
For RPN'en
1 23 + 45 6 + *
vil stakken være i følgende tilstander:
(), (1), (23 1), (24), (45 24), (6 45 24), (51 24), (1224).
```

Stakk-automaten som ASM: Universer og funksjoner

8.9 Statiske, dynamiske og eksterne funksjoner

Basisfunksjonene for en statisk algebra kan kategoriseres som statiske, dynamiske eller eksterne funksjoner:

- Statiske funksjoner er de basis funksjonene som ikke kan oppdateres under kjøringen av en ASM. Foreksempel så er funksjonene: Head, Tail, Push, Pop, Top statiske i stakk-automat eksempelet.
- Dynamiske funksjoner er de basis funksjonene som kan oppdateres under kjøring av ASMen. Fra Stakk-automaten har vi funksjonene Arg1, Arg2, S, S som er eksempler

```
universe Data, Oper, List, Stack
function Arg1 -> Data
function Arg2 -> Data
function Head:List -> DataUOper
function Tail:List -> List
function Apply: OperxDataxData -> Data
function Push:DataXStack -> Stack
function Pop:Stack -> Stack
function Top:Stack -> Data
function F: -> List
function S: -> Stack

if Head(F) is datum then
  S := Push(Head(F), S)
  F := Tail(F)
endif
if Head(F) is an operation then
 if Arg1 = undef then
     Arg1 := Top(S), S:= Pop(S)
 elseif Arg2 = undef then
     Arg2 := Top(S), S := Pop(S)
 else S := Push(Apply(Head(F), Arg1, Arg2),S)
      F := Tail(F), Arg1 = undef, Arg2 := undef
 endif
endif
```

Figur 8.8.1: ASM kode stakkautomat

på dynamiske funksjoner. Output oppførsel er modellert ved bruk av dynamiske basisfunksjoner. Feks. er det mulig at et bestemt element "Output" er satt til å skrive verdien sin til standardout etter hvert eksekveringssteg. Output := t

- Eksterne funksjoner brukes for å gjengi oppførsel fra "verden utenfor", e.g. bruker input, operativsystem, andre "utenforstående"ting. Eksterne funksjoner kan ikke endres av regler i ASMen. De kan være forskjellige for forskjellige tilstander til ASMen. En ekstern funksjon er et "orakel", en uforutsigbar black box som er brukt men ikke kontrollert av ASMen.

Vi forlater nå ASM og eksempler på ASM modeller og ser nærmere på bruken av ASM.

8.10 ASM i akademia

ASMer har fått en stor utbredelse innen akademia. Det finnes en egen nettside som prøver å samle alle de artikler som er skrevet om og som benytter seg av ASMer [5]. På denne siden kan en få et inntrykk av hvilke områder ASMer benyttes i og utbredelsen de har fått. I denne oppgaven ser vi først og fremst nærmere på de som har med protokoller å gjøre. Det er imidlertid et poeng å peke på det store mangfoldet vi finner i anvendelsesområder, noe som gjør ASMer til en allmenn spesifikasjonsmetode. Det er derimot anvendelsene i industrien som kanskje er mest interressante noe vi skal se på i neste seksjon..

8.11 ASM i næringslivet

ASMer har et stort bruksområde, og vi har hovedsakelig sett på de slik vi finner de i akademia. ASM som verktøy har derimot et stort bruksområde også i næringslivet. Den grunnleggende ideen var nemlig å utvikle et verktøy med solide teoretiske røtter i logikk og matematikk, samtidig som det skal kunne brukes av "ingeniører" uten alt for mye opplæring. For å bruke ASMer i utvikling av programvare og maskinvare måtte de innlemmes i hele prosessen. De måtte integreres på en måte som gjorde de praktiske i bruk og ikke som noe som skulle komme i tillegg til alt annet. Et eksempel på slik bruk er benyttelsen av ASMer i den s.k. "V-modellen" for programvareutvikling slik vi ser det i figur 8.11.1 side 68. Vi kan her se hvordan ASMer kan brukes først til en spesifikasjon som kan diskuteres med kunden. Deretter kan denne "grunnmodellen" gjøres mer detaljrik og også legge til rette for testing av moduler, funksjoner osv. i mot en implementasjon.

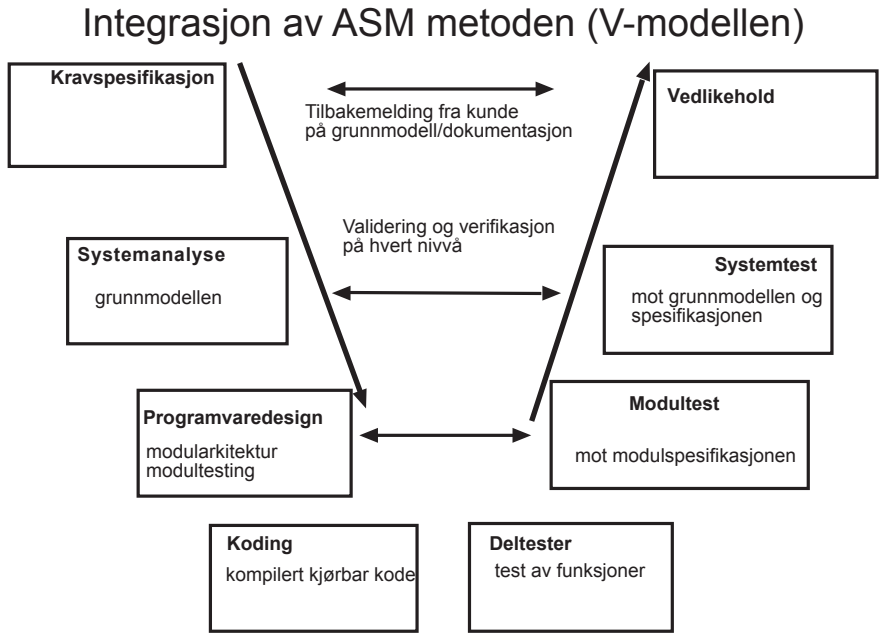
Bruken av ASM metodikken i industrien ble testet av Egon Börger under hans opphold hos Siemens i München fra 1998 til 1999. I hovedsak ble ASMer brukt i designmetoder for jernbanesystemer. Det ble utviklet verktøy for bruk i utvikling og testing av en modell for FALKO [6]. Metodene og verktøyene bl.a. ASM-Workbench ble videreutviklet for bruk også i andre prosjekter hos Siemens. Etter disse første anvendelsene av ASMer i industrien er de benyttet i mange ulike prosjekter hos forskjellige firmaer rundt om i verden. Noen av disse prosjektene er spesifikasjoner av programmeringsspråk; slik som Prolog og Java (Robert F. Stärk og Börger 2001). De er også benyttet til spesifikasjon av standarder; SDL-2000, maskinvare; DEC-Alpha prosessorer, arkitekturer; VHDL, PHDL osv.

8.11.1 Nyere industrielle anvendelser

Siden høsten 1998 har Yuri Gurevich vært tilknyttet Microsoft Research og det finnes flere eksempler på anvendelser av ASMer hos Microsoft etter dette. Blant anner et det laget

[5] Se http://www.eecs.umich.edu/gasm/

[6] FALKO - The FALKO® tool for design and validation of timetables was developed to optimise operational planning and control for operations control systems. Se www.siemens.com -> FALKO



Figur 8.11.1: Bruk av ASM i programvareutvikling

spesifikasjoner av Windows Card Runtime Environment, UPnP arkitekturen[7]. Det er også utviklet et språk for kjøring og testing av ASM spesifikasjoner kalt AsmL (Abstrakt tilstandsmaskinspråk), som er integrert i eksisterende Microsoft programvare. Det skal vi se nærmere på i kapittel 9. ASM spesifikasjoner er også benyttet i spesifisering og modellsjekking av et togstyringssystem i Australia og i arbeidet med utviklingen av sentrale funksjoner i et mobiltelefoni nettverk.

8.11.2 ASM verktøy

På grunn av bruken av ASM i industrien ble det raskt et behov for ulike verktøy for å kunne kjøre de abstrakte spesifikasjonene. Det er ikke nok bare å kunne kjøre simuleringer på ulike spesifikasjoner, det er også ønskelig å kunne benytte seg av standard verifikasjonsteknikker og testing og kunne gjøre forbedrelser, versjonskontroll og vedlikehold.

ASM-Interpreter Vi har en lang rekke av interpreter for ASMer. Noen av de første ble utviklet under arbeidet med en ISO–standard for programmeringsspråket Prolog. Det ble bl.a utviklet en interpret for sekvensielle ASMer i Prolog og det ble også utviklet og implementert en funksjonell interpret for ASM i Scheme, i Oslo (Diesen 1995). Det ble videre utviklet en interpret ved universitetet i Paderborn kalt ASM Workbench som bl.a. ble benyttet i FALKO prosjektet hos Siemens. Bruken av Interpreter i ASM sammenheng gjorde det mye enklere å se hva som skjer i en kjøring av programmer. Slik er det mulig å stadig teste modellen med hensyn til ulike scenarier i kjøringen av programmet. Ved utviklingen av ASM–interpreter ble det også naturlig å se nærmere på ASM og bruken av verifiseringsverktøy.

ASM og verifiseringsverktøy ASM er brukt sammen med teorembevisere som Isabelle, PVS og KIV. De er også benyttet sammen med modelsjekkere som MDG, SMV m.fl. Det er også utviklet flere logikker og verktøy for mekanisk resonnering for å studere og arbeide med ASMer (Börger og Stärk 2003).

8.12 ASM oppsumering

Vi har sett på ASMer og deres bruksområder som spesifikasjonsmetode og verktøy. Vi har sett på bruken av de i utviklingen av programmer, protokoller og maskinvare. Det er gjort ved både å se på det matematiske fundamentet slik vi finner det hos (Börger og Stärk 2003; Gurevich 1994) og i tillegg har vi sett på eksempler på ASMer hentet fra (Anlauff 1998; Gurevich 1993; Huggins og Wallace 2002). I neste kapittel går vi over til å se på et kjørbart spesifikasjonsspråk som bygger på teorien om abstrakte tilstandsmaskiner. Vi vil så bruke dette spesifikasjonsspråket til å lage en modell for Needham-Schroeder protokollen for autentisering.

[7] Universal Plug and Play - for å kunne koble til og kommunisere med ulike "smarte" enheter automatisk.

Kapittel 9

AsmL - Abstrakt tilstandsmaskinspråk

9.1 AsmL innledning

I forrige kapittel så vi på ASMer og bruken av de. I dette kapittelet skal vi se nærmere på AsmL, Abstrakt tilstandsmaskinspråk. Språket er utviklet ved Microsoft Research av bl.a Yuri Gurevich. AsmL blir beskrevet som et kjørbart spesifikasjonsspråk som bygger på teorien om Abstrakte tilstandsmaskiner (Microsoft Research). Hos Microsoft er AsmL i bruk i utviklingen av programvare og/eller maskinvare. De beskriver det som et godt verktøy for å gi nøyaktige ikke-tvetydige spesifikasjoner som kan brukes av ulike utviklingsgrupper. AsmL kan kjøres, noe som gjør utviklerne i stand til å utforske designet som de jobber med. Fungerer alt slik du har tenkt det? Er det noen uforutsette effekter? Er det andre problemer som dukker opp i arbeidet med spesifikasjonen? Slike spørsmål er naturlig å stille seg i arbeidet med spesifikasjonen. Siden AsmL spesifikasjonen er kjørbar er det enkelt å spore forandringer i spesifikasjonen, for å se hvilke virkninger de får. Det kan være nyttig å jobbe med spesifikasjonen for å teste den godt før en går i gang med eventuell koding. Vi har beskrevet noe av arbeidet med Asml i seksjon 8.11.1. Det er et arbeid som fortsatt pågår hos Microsoft (bl.a. med å bruke AsmL for å teste komponenter i .NET arkitekturen til Microsoft (Barnett og Schulte 2003)). Det er også mulig å bruke AsmL til runtime verifisering, og det er utviklet verktøy for å teste en spesifikasjon i "runtime" imot implementasjonen, ved hjelp av en parallell kjøring.

9.2 AsmL systemet

Selve AsmL systemet er en utvidelse av Microsoft Visual Studio .Net og Microsoft Word. AsmL spesifikasjonene ser like ut i begge omgivelsene, selv om det kanskje virker mest naturlig å benytte MS Visual Studio og de debug muligheter som finnes der. AsmL benytter seg av .Net rammeverket som er en Windows–komponent for å utvikle og kjøre

programvare[1]. Rammeverket inneholder en felles "language runtime" og et klassebibliotek. Det støtter over 20 språk blant andre: C#, C++, J#, Java, Eiffel, Fortran, Haskell, Scheme, Perl, Python, Standard ML, Pascal Smalltalk m.fl. AsmL bruker .Net rammeverket og AsmL koden blir kompilert til C# kode for kjøring. Det gjør at det er mulig å lage AsmL spesifikasjoner ved hjelp av kun Word og .Net rammeverket sammen med AsmL programmet. Både .Net rammeverket og AsmL er gratis å bruke. Uansett om en velger å bruke Word eller Visual Studio skrives AsmL spesifikasjonen som et XML-dokument. Det er egne XML tagger for kode og kommentarer. Dette XML dokumentet kan vises frem i en nettleser og vi får presentert kode og kommentarer på en lettlest og god måte (se figur A.14.1 i tillegg A). Det er kun selve kodebitene som blir kompilert til C# og som kjøres. Resultatet av kjøringene dukker opp i et konsollvindu (se figur A.13.1 i tillegg A). Det at kode og kommentarer er vevet sammen i XML dokumentet gjør at dokumentet kan leses som s.k. "literate programming" jf. Donald Knuth (Knuth 1984):

"I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature. Hence, my title: "Literate Programming." Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do. The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that reinforce each other."

Dette gjør at en hele tiden er oppmerksom på å gi koden de nødvendige kommentarer samtidig som selve AsmL språket er lettfattelig og har mange elementer til felles med kjente programmeringsspråk. AsmL benytter seg av objektorientering slik vi kjenner den fra Java, om enn litt enklere.

[1].Net forstås generelt hos Microsoft som utviklingsverktøyene, webservices, servere og .Net rammeverket.

9.3 AsmL eksempel

Et kort eksempel på en AsmL spesifikasjon kan være "Shortest Path", som kommer med som eksempel når du installerer AsmL. Spesifikasjonen består av en algoritme for å finne korteste vei i en graf til en gitt node s. Nodene til grafen er gitt som en mengde N, og avstanden mellom nodene er gitt som en mapping D. Med D(n,m) = infinity mener vi at to noder ikke er tilstøtende. Selve spesifikasjonen ser da ut som i 9.3.1:

```
shortest( s as Node, N as Set of Node,
          D as Map of (Node, Node) to Integer) as Map of Node to Integer
  var S = {s -> 0} + {n -> infinity | n in N where n <> s}
  step until fixpoint
    forall n in N where n <> s
      S(n) := min S(m) + D(m,n) | m in N
  step
    return S}
```

Figur 9.3.1: AsmL : Shortest path algoritme

For å gjøre spesifikasjonen komplett og kjørbar definerer vi klassen *Node*, som kun består av en unik string:

```
structure Node
  name as String
```

Vi vet at uendelig er et stort tall, i vår sammenheng holder det med:

```
inifinity = 9999
```

Vi avslutter spesifikasjonen med kjøringen av et eksempel. Her er det snakk om to byer som ikke har noen direkte forbindelse, og vi kan derfor bruke "shortest path" algoritmen. For å utføre kjøringen er det nødvendig med en Main() hvor vi definerer Nodene, mappingen og sier hvilke noder (byer) vi skal regne ut avstanden mellom se 9.3.2.

```
Main()
  SeaTac = Node("SeaTac")
  Seattle = Node("Seattle")
  Bellevue = Node("Bellevue")
  Redmond = Node("Redmond")
  N = {SeaTac, Seattle, Bellevue, Redmond}
  D = {(SeaTac, SeaTac) -> 0,
       (SeaTac, Seattle) -> 11,
       (SeaTac, Bellevue) -> 13,
       (SeaTac, Redmond) -> infinity, // to be calculated
       (Seattle, SeaTac) -> 11,
       (Seattle, Seattle) -> 0,
       (Seattle, Bellevue) -> 5,
       (Seattle, Redmond) -> 9,
       (Bellevue, SeaTac) -> 13,
       (Bellevue, Seattle) -> 5,
       (Bellevue, Bellevue) -> 0,
       (Bellevue, Redmond) -> 5,
       (Redmond, SeaTac) -> infinity, // to be calculated
       (Redmond, Seattle) -> 9,
       (Redmond, Bellevue) -> 5,
       (Redmond, Redmond) -> 0}
  shortestPathsFromSeaTac = shortest(SeaTac, N, D)
  WriteLine("The shortest distance from SeaTac to Redmond is " +
            shortestPathsFromSeaTac(Redmond) +
            " miles.")
```

Figur 9.3.2: AsmL : Main()

Kjøring av eksempelet gir:

```
The shortest distance from SeaTac to Redmond is 18 miles.
```

Dette er en veldig enkel spesifikasjon, men den er et godt eksempel på hvordan AsmL kode skrives, eksekveres og hvor lettlest slik kode kan være.

Kapittel 10

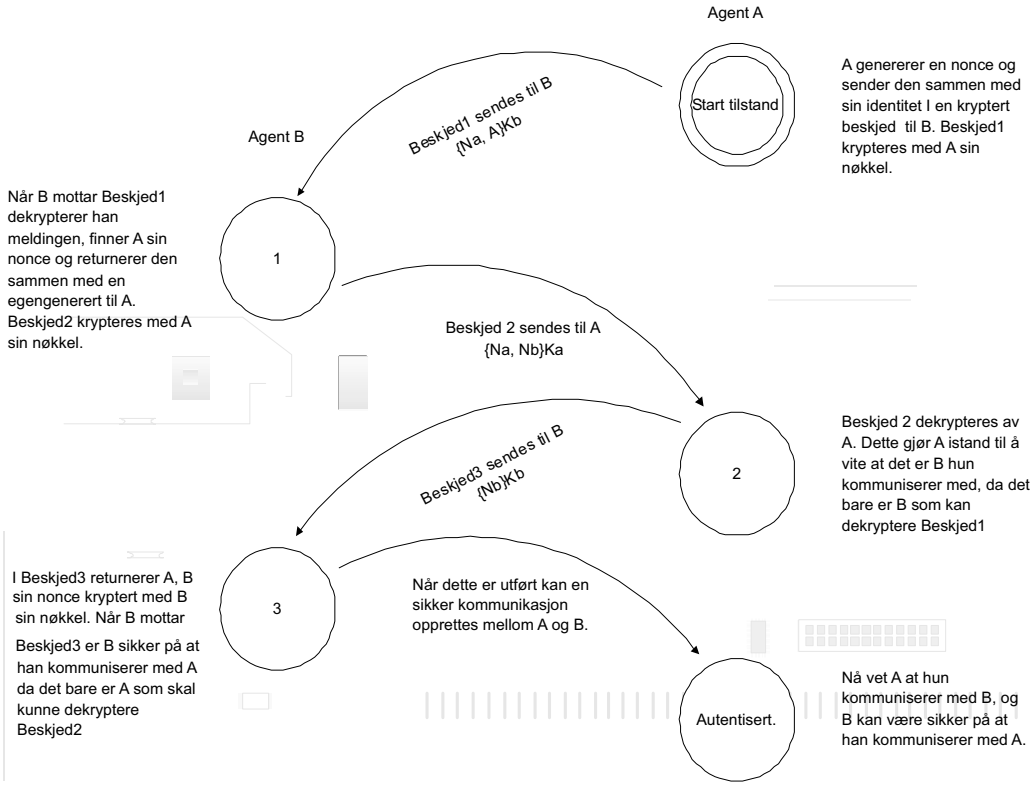
En AsmL spesifikasjon av Needham-Schroeder protokollen

Vi går nå over til å se nærmere på en AsmL spesifikasjon av Needham-Schroeder protokollen for autentisering som er utviklet i arbeidet med denne oppgaven. Modellen er laget i AsmL og arbeidet med den viser en spesifikasjon av en enkel variant av protokollen. I kapittel 7 så vi på Needham-Schroeder protokollen for autentisering som benytter seg av asymmetrisk kryptografi. I 7.1.1 ser vi den fullstendige protokollen hvor to agenter A(lice) og B(ob) autentiserer hverandre etter å ha hentet offentlige nøkler fra en nøkkelserver S som de begge tidligere har utvekslet nøkler med. Vi skal her se nærmere på hvordan selve autentiseringen foregår og vil derfor se på meldingene 3, 6 og 7 fra del 7.1.1 side 48. Vi forutsetter at agentene A og B har tilegnet seg de riktige offentlige nøklene til hverandre fra en nøkkelserver S. Vi får da følgende protokoll (samme som i del 7.2.1 side 49):

1. A $\rightarrow$ B : $\{N_a, A\}_{Kb}$
2. B $\rightarrow$ A : $\{N_a, N_b\}_{Ka}$
3. A $\rightarrow$ B : $\{N_b\}_{Kb}$

Figur 10.0.1: Meldinger som kun tar for seg autentisering.

Som i har sett tidligere er det mye som skjer mellom linjene her. Vi kan derfor tenke oss en skjematisk framstilling av hvordan protokollen kan beskrives som tilstander og transisjoner mellom de (se figur 10.0.2). Dette for å få en visualisering av protokollen som vi kan bruke i arbeidet med å utvikle en modell. Merk at tallene i figur 10.0.2ikke tilsvarer reelle tilstander.



Figur 10.0.2: Figur av N-S AsmL modellen 1

10.1 Needham-Schroeder modellen

I modellen kan vi tenke oss at tilstandene blir oppdatert i hvert enkelt objekt av typen *Agent* og at transisjonene skjer ved hjelp av stegene i AsmL. Et steg i en AsmL kan sammenlignes med et steg i en ASM. Hvert steg kan sees på som overgangen fra en tilstand til en annen. En måte AsmL kan brukes til å spesifisere sekvensielle algoritmer er ved hjelp av formen (step ... step) som indikerer en sekvens av steg som eksekveres etter hverandre dvs. sekvensielt. Dette kommer klarere frem i *Main()* av modellen se tillegg (A). Dersom vi ser på figur 10.0.2 vil vi se at vi trenger følgende elementer:

10.1.1 Elementene i modellen

- Agent - et agentobjekt som inneholder de nødvendige variable.
- Beskjed - en beskjed inneholder nonce, identitet og er kryptert med en nøkkel.
- Nonce - en randomgenerert verdi.
- Kryptering - en funksjon som sjekker om krypteringen er riktig.
- Nøkkel - nøkkelen som er benyttet for å kryptere beskjeden.

Disse elementene danner en basis for å skrive en spesifikasjon av protokollen. Siden jeg tar for meg en veldig enkel versjon av protokollen er det noen elementer jeg ikke trenger, men som kan være nødvendige i utvidelser av den. Dette er elementer som *tidsstempel*, *serveragent* m.m. Hele spesifikasjonen med AsmL–kode og kommentarer finnes i tillegg A.

10.1.1.1 Agent

Vi starter med å se nærmere på klassen for *Agent*–objektet. Det er definert i koden som :

```
public class Agent
  var navn as String
  var nøkler as Map of String to Integer
  var tilstand as String
  var mailbox as Seq of Beskjed = []
  var nonces as Seq of Nonce = []
```

Figur 10.1.1: Agent–klassen

En agent representerer en bruker eller et program som skal operere i et datanettverk. Agenten skal kunne opprette sikre kommunikasjonskanaler med andre agenter ved hjelp av en autentiseringsprotkoll. Etter at autentisering mellom partene er foretatt, er det mulig å utveksle symmetriske nøkler for kryptert kommunikasjon (se del 3.5.3 side 19). Agenten

består i modellen av en klasse med følgende felter: *navn* (identitet), *nøkler* (de offentlige nøklene agenten har), tilstand (for å angi hvilken tilstand agenten befinner deg i til enhver tid), *nonces* (hvilke nonces agenten har mottatt) og en *mailbox* hvor beskjeder fra andre agenter kommer inn. Som vi kan se er de forskjellige variablene deklarert vha. ***var*** *x* ***as*** *y* hvor *y* er de ulike innebygde typene i AsmL. Vi kommer tilbake til noen av de forskjellige typene, men de er hovedsakelig slik vi kjenner de fra andre programmeringsspråk.

10.1.1.2 Nonce

Nonce[1] er en tilfeldig verdi (random) som kun skal brukes en gang. Nonce blir generert av agentene i protokollen. Nonce er realisert som en klasse med en verdi; et randomgenerert tall, og en identitetsvariabel som angir hvilken agent som genererte verdien. Nonce benytter seg av *structure* som i AsmL er en sammensatt verdi. Det at *structure* her er *sealed* vil si at metoder som bruker *Nonce* ikke kan gjøre forandringer på den.

```
sealed public structure Nonce
  verdi as Integer
  ident as String
```

Figur 10.1.2: Nonce structure

Når en agent har mottatt en nonce, blir denne lagret i agentens variabel *nonces* som en sekvens. Agenten må sjekke at den nonce som en agent får tilbake i neste melding fra motparten er den samme som agenten sendte fra seg. Vi kan se i figur 10.0.1 at agent A sender en nonce til B i beskjed 1 og skal motta *den samme* nonce fra B i beskjed 2. Det samme gjør B ved å sende sin nonce i beskjed 2 og motta denne igjen i beskjed 3. Å sende, for så å motta samme *nonce* er avgjørende for selve autentiseringen.

I modellen skjer dette ved hjelp av funksjonen *SjekkNonce* som sjekker om nonce mottatt, er den samme som den som ble sendt av samme agent i forrige beskjed. Funksjonen gjør også en sjekk av identiteten til den som genererte noncen da *Id* er en del av *Nonce* "structuren".

```
function SjekkNonce (a as Nonce, b as Nonce)as Boolean
  if b = a then
    true
  else
    false
```

Figur 10.1.3: Funksjonen SjekkNonce

[1] "Nonce word" - et "ord" som finnes eller er opprettet for å brukes kun en gang. Fra Encyclopædia Britannica Online. <http://search.eb.com/eb/article?eu=137930>

10.1.1.3 Nøkler og kryptering

Nøkler er implementert som klassen *Nøkkel*. Generelt vil vi bruke offentlige nøkler (ON) og private nøkler (PN). I arbeidet med protokollen forutsetter vi at kryptering og dekryptering foregår på en sikker og god måte. Det finnes flere algoritmer som gjør det, bl.a. RSA algoritmen som vi så i del 3.5.2. En implementasjon av en krypteringsalgoritme vil være utenfor rekkevidden av denne oppgaven. Det som derimot er viktig, er at nøkklene blir behandlet og brukt på en korrekt måte. Vi forutsetter at de offentlige og private nøklene er utvekslet på en riktig måte slik vi ser det i delen om Needham-Schroeder protokollen kapittel 7 side 47. For arbeidet med modellen er det derfor kun nødvendig å sjekke om en agent har tilgang til riktig nøkkel for å kunne dekryptere beskjeder han eller hun mottar. Vi sjekker om riktig nøkkel er benyttet ved hjelp av funksjonen *Kryptering* (se figur 10.1.4 side 79).

```
function Kryptering(i as beskjed, j as String) as Boolean
  if j in i.nøkkel then
    true
  else
    false
```

Figur 10.1.4: Funksjon for å sjekke kryptering

Kryptering vil sikre at agenten innehar riktig nøkkel for å kunne lese beskjeden den har mottatt. Dersom agenten ikke har riktig nøkkel, vil en eventuell kjøring stoppe og gi en feilmelding. Det signaliseres med bruk av *error* og teksten "Feil nøkkel". En slik bruk av *error* vil sørge for at kjøringen av programmet stopper umiddelbart.

Bruk av kryptering fra *Main()*:

```
if Kryptering(mb, "OnB") then       //sjekker om kryptering er gjort med riktig nøkkel
   WriteLine("Kryptering OK") else
   error "Feil nøkkel"
```

Figur 10.1.5: Sjekk av kryptering i Main()

10.1.1.4 Beskjed

Beskjed består av de feltene som er nødvendige for å utføre en autentisering basert på utveksling av krypterte beskjeder . *Beskjed* inneholder variablene *nonce1*, *nonce2*, *id* og *nøkkel*. Hvor *nonce1* og *2* er respektive nonces, i noen beskjeder vil det bare være en nonce. *Id* er identiteten til den av agentene f.eks. *Alice* eller *Bob* som sender beskjeden. Nøkkel er den (offentlige) nøkkelen som er brukt for å kryptere beskjeden. Dette siste er viktig, da vi forutsetter at en agent har brukt *riktig nøkkel* for å kryptere beskjeden, hvis ikke vil kjøringen av modellen stoppe når mottager av meldingen sjekker krypteringen.

```
public class Beskjed
  var nonce1 as Nonce
  var nonce2 as Nonce or Null
  var id as String
  var nøkkel as Set of String
```

Figur 10.1.6: Beskjed klassen

10.1.1.5 Kjøring av spesifikasjonen

Vi har også noen ekstra funksjoner for å hjelpe oss i kjøringen av spesifikasjonen. *Oppdater-Tilstand* er en funksjon som avhengig av tilstanden som en agent befinner seg i oppdaterer tilstanden til agenten. Her er tilstand å forstå som variabelen *tilstand* som tilhører en gitt agent. Jeg har kun tatt med akkurat de tilstandene en agent må gå igjennom for å bli autentisert (se figur 10.1.8 side 82. I selve kjøringen sjekkes *tilstand*, og en agent vil ikke gå til neste tilstand uten at alt er slik det skal være i forrige tilstand.

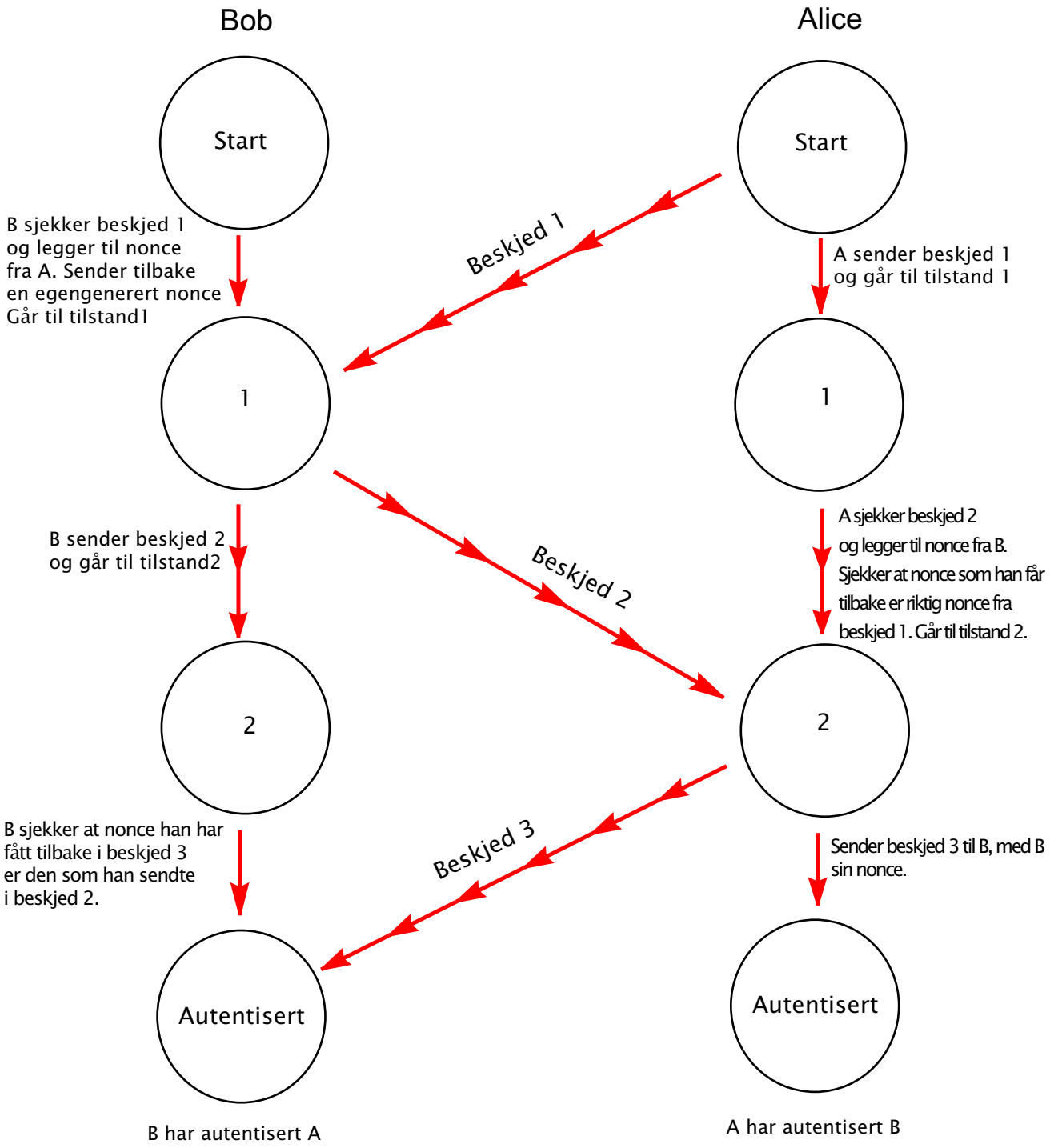
- Kryptering må være gjort med riktig offentlig nøkkel.
- Eventuelt nonce lagt til hos agent.
- Beskjed sendt eller mottatt.

LeggTilNonce vil legge til en nonce som er mottatt i en beskjed i *nonces* variabelen hos agenten. Det gjøres etter at det er utført en sjekk på om meldingen er kryptert riktig.

```
function OppdaterTilstand (a as Agent, t as String)
   if t = "start" then
     a.tilstand := "1"
   elseif
     a.tilstand = "1" then
     a.tilstand := "2"
   elseif
     a.tilstand = "2" then
     a.tilstand := "Autentisert"


function LeggTilNonce (a as Agent, n as Nonce)
   a.nonces := [n]
```

Figur 10.1.7: Funksjonene OppdaterTilstand og LeggTilNonce

For å få et riktig bilde av når nonce legges til, kryptering sjekkes, beskjed sendes og tilstand oppdateres kan vi se figur 10.1.8 side 82.



Figur 10.1.8: Figur av N-S AsmL modellen 2

10.1.2 Kommunikasjon mellom agentene

Vi behøver funksjoner som hjelper oss med å få agentene som blir opprettet til å kommunisere. Kommunikasjonen foregår ved utveksling av beskjeder. Disse beskjedene er de samme som de vi har definert i klassen *Beskjed* (se figur 10.1.6). Jeg prøver her å bruke noe av koden som er skissert hos Uwe Glässer og Veanes (2002). De tenker seg at hver agent har en mailbox som de andre agentene kan putte meldinger i. Jeg legger til den nødvendige mailboxen for dette i Agent–klassen. I tillegg benytter jeg meg av funksjonen *SendMessage*, den putter beskjeden fra en agent inn i mailboxen til mottageragenten som spesifiseres i funksjonskallet.

```
SendMessage(a as Agent, m as Beskjed)
        a.mailbox := [m]
```

Figur 10.1.9: Funksjonen SendMessage

10.1.3 Kjøring av modellen

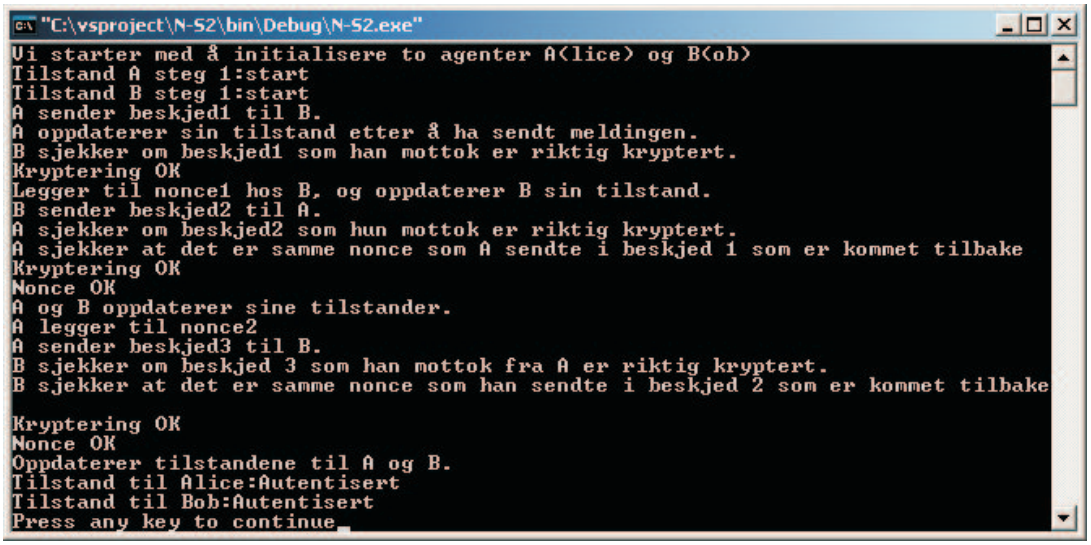
Her kommer selve eksekveringen av modellen. Her benyttes de ulike klasser og funksjoner som er deklarert ovenfor for å realisere en eksekvering av modellen. Modellen starter med at to agenter Alice og Bob initialiseres og videre ser vi hvordan de kan gjennomføre en autentisering ved hjelp av protokollen som modellen realiserer. Selve kjøringen skjer i en *Main* del, der benyttes *steg* i stor grad for å få utført de aktuelle operasjoner samtidig før neste tilstand. Slik som vi ser under i utdrag fra *Main()* i tillegg A:

```
[EntryPoint]
Main()
  var a = new Agent("Alice",{"On(B)" -> 1},"start")
                    //Initialiserer to agenter Alice og Bob.
  var b = new Agent("Bob",{"On(A)" -> 2},"start")
  var beskjed1 = new Beskjed(Nonce(GetRandomSeed(), "A"),null,"A
      ",{"OnB"})

  step
    WriteLine("Vi starter med å initialisere to agenter A(lice)
        og B(ob)")
    WriteLine("Tilstand A steg 1:" + a.tilstand)
    WriteLine("Tilstand B steg 1:" + b.tilstand)
    WriteLine("A sender beskjed1 til B.")
    SendMessage(b, beskjed1)                          //Sender
        beskjed1 til Bob fra Alice

    OppdaterTilstand(a, "start")                     //Oppdaterer
         Alice sin tilstand fra start til 1
    WriteLine("A oppdaterer sin tilstand etter å ha sendt
        meldingen.")
```

Figur 10.1.10: Utdrag fra Main()

I figur 10.1.10 ser vi hvordan vi i linje 4-6 oppretter agenter og den første beskjeden i protokollen som sendes av A. Etter *step* i linje 7, ser vi hvordan A sender beskjed1 til B og oppdaterer sin egen tilstand. Slik fortsetter protokollen igjennom alle tilstander som er nødvendige for å fullføre autentiseringen.

Resultatet av kjøringen dukker opp i et DOS-vindu hvor vi kan se resultatet av hva som skjer. For å få et slikt kjøringseksempel må vi legge til en del *WriteLines* som skriver til skjermen hva som skjer. Dette ser vi i figur 10.1.11.



Figur 10.1.11: Kjøringseksempel for Needham-Schroeder i AsmL.

10.1.4 Hva har vi lært av arbeidet med modellen?

Modellen er en spesifikasjon av Needham-Schroeder protokollen slik vi finner den i sin enkleste variant. En slik spesifikasjon eller modell av en protokoll er en veldig god dokumentasjon av hva som er formålet med protokollen, og hvordan den er tenkt realisert. Vi kan også utsette mange valg inntil vi er sikre på å ha en protokoll som er logisk riktig. Enkelte valg er knyttet direkte til selve implementasjonen og kan derfor utsettes til vi går løs på implementeringen. Arbeidet med Needham-Schroeder protokollen har vist at den enkle protokollen som vi har sett på, ikke kan motstå s.k. "man in the middle" angrep (se kap. 7 del 7.2). En løsning på akkurat dette problemet er å være mer presis med å spesifisere hvem som er mottaker og avsender av en melding. Det kommer også klart frem i flere artikler som viser feil i protokoller på grunn av mangel på klarhet når det gjelder sender og/eller mottaker av en melding. Vi kan se dette hos Anderson og Needham (1995), og hos Bella (2001) som begge tar opp dette.

Et annet problem kan være hvordan kommunikasjonsnøkler behandles. Det ser vi i kap. 7 del 7.3.1. En løsning på problemet er å introdusere tidsstempler, men bruk av tidsstempler introduserer nye problemer i selve implementasjonen. Alle valg vi gjør påvirker de forutsetningene vi hadde innledningsvis. Tidsstempler forutsetter at nettverket har synkroniserte klokker med god nøyaktighet, dersom tidsstemplene skal kunne brukes på en effektiv måte. Bruken av tidsstempler er derfor underlagt forutsetninger som kan være andre enn de vi startet med. Vi kan tenke oss at autentiseringen foregår asynkront og at hver beskjed bruker en viss tid både på å komme frem, og på å bli besvart. Med disse forutsetningene vil kanskje ikke tidsstempler være hensiktsmessige.

Kommunikasjon er et annet område hvor det kan være vanskelig å gjøre de riktige forutsetningene. Vi vil kanskje forutsette at alle meldinger som blir sendt, kommer fram til riktig mottaker, i riktig rekkefølge, innenfor en rimelig tid. Det er ikke alltid slik. Her kommer også angriperen inn, slik vi så i kapittel 5 del 5.4. Dersom vi forutsetter at angriperen har full tilgang til nettverk og meldinger slik Dolev og Yao tenker seg, må en utvide spesifikasjon til å også ta hensyn til dette, se del 10.2 side 88.

AsmL gjør oss i stand til å tenke igjennom spørsmål slik som de ovenfor, og vi kan teste ulike løsninger direkte, ved å legge de til i modellen og kjøre den på nytt. Slik kan vi stadig forbedre AsmL–koden til vi kommer til et punkt hvor vi er klare til å implementere. Modellen kan så utnyttes i arbeidet med implementasjonen som et "orakel" ved å vise hvilke datastrukturer som kan brukes, og hvordan funksjonene bør se ut. Modellen kan kjøres parallelt med en implementasjon slik at vi ser at de riktige kall og oppdateringer skjer samtidig. Vi kan koble sammen flere og flere komponenter til vi får en fullt ut fungerende protokoll. Komponenter som vi vil måtte legge til i modellen vi har sett på av Needham-Schroeder er f.eks. komponenter som gir oss de krypteringsalgoritmene som vi tenker å bruke. Arbeidet og bruken av AsmL spesifikasjonen vil ikke stoppe med en ferdig utviklet protokoll, vi får også bruk for den senere, noe vi ser mer på i del 10.2.

10.1.5 Sterke og svake sider ved AsmL metoden

Slik vi har sett med de ulike formelle metodene i kapittel 6, har også AsmL sine sterke og svake sider. AsmL er en generell metode som er utviklet til å se på ulike former for protokoller, programvare, maskinvare og systemer. Det store bruksområdet får følger når det gjelder hvilke svakheter som kan avdekkes og hvor enkelt det er å finne disse. Noen av de andre formelle metodene vi har sett på, er spesialutviklet for å behandle autentiseringsprotokoller, og disse kan derfor forventes å avdekke flere feil ved eksisterende protokoller enn det AsmL vil gjøre. Grunnen til dette er at mange av de spesialutviklede metodene behandler momenter som er spesifikke for akkurat autentiseringsprotokoller på en bedre måte enn en generell metode, som AsmL. Det er også utviklet spesialsystemer for enkelt protokoller, disse er selvfølgelig veldig gode på akkurat "sin" protokoll. AsmL, som en generell metode, er kanskje ikke den best egnede til analyse og feilretting av eksisterende autentiseringsprotkoller, den har sin styrke i designarbeid. Ved bruk av AsmL har vi et verktøy som egner seg godt til bruk i hele designprosessen, og som kan gjøre feilretting og vedlikehold enklere i utviklingen av nye autentiseringsprotokoller.

10.1.6 Erfaringer med ASM og AsmL verktøy

AsmL kan sees på som et hvilket som helst annet programmeringsspråk, og det benytter seg av kjente konstruksjoner og datatyper som vi kjenner fra andre programmeringsspråk. Å lære seg AsmL er derfor omtrent som å lære å bruke andre programmeringsspråk. Per i dag finnes det ingen lærebøker i bruk av AsmL, det er derfor nødvendig med en del prøving og feiling i starten. Det finnes en "tutorial" (Foundations of Software Engineering Microsoft Research b) og en manual (Foundations of Software Engineering Microsoft Research a) som følger med nedlastingen av AsmL programmet. I tillegg finnes det et lite, men aktivt miljø verden over som benytter seg av AsmL. Dette miljøet er en gruppe som omfatter både akademikere og personer i næringslivet bl.a. hos Microsoft. Det finnes en egen epostliste dersom en ønsker å stille spørsmål eller komme med "ønsker" til utviklerne av AsmL hos Microsoft Research [2].

Forholdet mellom ASM og AsmL Det er kanskje nærliggende å tro at overgangen fra ASM til AsmL er uten problemer, det er dessverre ikke slik. Det finnes ingen automatisert overgang fra en ASM, til en AsmL spesifikasjon av samme problem. Det kan likevel være gunstig å starte med en enkel ASM før en går løs på AsmL programmeringen. Igjennom arbeidet med en ASM har en gjort en del tanker om hvilke funksjoner, datatyper og strukturer som er nødvendig for å spesifisere protokollen eller algoritmen som det arbeides med (se eksempelet på en ASM del 8.7 side 62). Prosessen med å finne de ulike "universer" til ASMen, kan sammenlignes med å finne passende datastrukturer i AsmL. Funksjonene vil også ligne på hvernadre. Grunnen til at vi kanskje foretrekker AsmL er at de deler presise matematiske definisjoner med ASM, samtidig som en AsmL spesifikasjon kan kjøres.

[2] Adressen er: asml@list.research.microsoft.com

Riktignok finnes det flere interpretere og omgivelser for "å kjøre" ASMer. AsmL er kanskje å foretrekke da den er mest brukt og under kontinuerlig utvikling, det finnes også støtte for modellsjekking se del 10.2.1.

10.2 Videre arbeid

Arbeidet i oppgaven kan utvides i flere retninger. Vi vil her se nærmere på noen av disse, spesielt de som er knyttet til AsmL spesifikasjon av autentiseringsprotokoller. Vi starter med å se på spesifikasjonen slik vi finner den i del 10.1 side 77. Det første som kan gjøres er å utvide denne spesifikasjonen til å omfatte en Needham-Schroeder protokoll som benytter seg av en nøkkelserver S, vi må da legge til en slik serverfunksjon som holder orden på offentlige nøkler og distribuerer disse til agentene A og B slik vi finner det hos Needham og Schroeder (1978a). Videre vil det være naturlig å ta fatt i kritikken som er lagt frem av Denning og Sacco (1981) og Lowe (1995). Dette gjør at vi må legge til tidsstempler slik som foreslått av Denning og Sacco. Med tidsstempler får vi noen nye forutsetninger, nemlig hvordan kommunikasjonen skal foregå i systemet for å sikre god nok sikkerhet i forhold til synkrone klokker osv. Det er også naturlig å legge inn identiteten til avsender i beskjed 2, slik at vi unngår et "man in the middle" angrep slik vi finner det hos Lowe 1995. Det er også en del arbeid som gjenstår i forbindelse med typesjekking. Typesjekking kan f.eks. hindre agenten i å sende en beskjed uten å kryptere den først eller andre slike typefeil. Vi sjekker at beskjeden er kryptert før den blir sendt osv.

Kryptering For å få en bedre behandling av kryptering enn det som er gjort i spesifikasjonen som vi har sett på, er det mulig å legge til egne funksjoner og strukturer for det slik som vi finner det i (Rosenzweig, Runje, og Slani 2003). I denne artikkelen med tilhørende AsmL kode på nettet [3], finner vi en kjørbar formell modell av abstrakt kryptering. Modellen oppretter klasser for offentlige og private nøkler, som brukes sammen med s.k. *beskjedmønstre* for å studere kryptografiske protokoller. Det er mulig for hånd å komme frem til kjente "sikkerhetsbevis" for noen enkle protokoller (f.eks. Needham-Schroeder). Dette er en retning som med utvidelser kan gi maskinstøttet, eller til og med automatisk bevis av sikkerhetsegenskaper ved autentiseringsprotokoller brukt i næringslivet.

Kommunikasjon Kommunikasjon mellom agenter er et annet veldig interessant moment som kan behandles på ulike måter. En god behandling av kommunikasjon er nyttig uansett hvilken protokoll vi vil studere. En mulighet er å utvikle et kommunikasjonsnettverk slik som vi finner det hos Glässer, Gurevich, og Veanes (2002). Her finner vi en abstrakt modell av et datanettverk med agenter som kommuniserer med hverandre. I modellen av Needham-Schroeder vi har sett på i del 10.1, har vi benyttet oss av deler av denne. Det er ikke noe i veien for å utvide vår modell til også å modellere et datanettverk med agenter slik som i

[3] Artikkel med tilhørende AsmL kode kan finnes på http://www.fsb.hr/ drosenzw/protocols/ (per 8.12.03)

artikkelen til Glässer et al. 2002. Med en slik modell av et datanettverk er det også naturlig å se nærmere på hvordan vi kan gå fram for å modellere en angriper.

Angriper Det kan være interessant å formalisere en angriper i AsmL. Det kan gjøres på flere måter. I artikkelen til Bella og Riccobene (1998) ser vi et eksempel på hvordan de tenker seg en angriper med full kontroll over et datanettverk er formalisert som en ASM. Denne ASMen kan brukes som bakgrunn for å innføre en angriper i AsmL modellen vår. En slik angriper vil være nyttig i alt arbeid med sikkerhetsprotokoller, ikke bare autentiseringsprotokoller.

```
SPY-ILLEGAL
R) do in-parallell
     add fake info to traffic
     save learned nonces
     destroy info from traffic
   end-par
```

Figur 10.2.1: Angriper i ASM

Som vi ser av figur 10.2.1 kan denne angriperen både gjøre endringer i trafikken, ødelegge trafikk, sende trafikk og lære seg noncer sendt på datanettverket. Dette er en inntrenger med sterkere egenskaper en den vi så hos Dolev og Yao i del 5.4 på side 38.

Andre protokoller og systemer Needham-Schroeder protokollen er veldig gammel i "næringslivsforstand". Den er likevel mye brukt i akademia som utgangspunkt for analysearbeid og metodeutvikling. Protkollen inneholder få meldinger og var utgangspunktet for de fleste andre autentiseringsprotokoller. Ingen implementasjoner bruker Needham-Schroeder protokollen for autentisering i sin originale form nå i dag. Men selve ideene bak Needham-Schroeder brukes den dag i dag og vi finner blant annet en implementasjon av en variant av den i autentiseringssystemet Kerberos [4]. Kerberos benytter seg av Needham-Schroeder protokollen med tidsstempler, og kun private nøkler som distribueres av en nøkkelserver som grunnlag for sitt system. En AsmL modell som spesifiserer Kerberos kunne derfor være veldig interessant. En ASM spesifikasjon av Kerberos finnes i Bella og Riccobene (1997). De kommer frem til en komplett analyse av Kerberos ved hjelp av stegvise forbedringer slik vi kjenner de fra del 8.11 på side 67. Modellen av Kerberos slik vi finner den hos Bella og Riccobene vil være et godt utgangspunkt for en AsmL–modell. Kerberos er bare et eksempel på en protokoll som kan være et godt utgangspunkt for en AsmL–modell.

[4]Kerberos bygger på Needham-Schroeder protokollen. Kerberos har sitt utspring fra project Athena hos MIT, som skulle sørge for autentisering i et distribuert datanettverk. Vi finner Kerberos i mange ulike applikasjoner og systemer som brukes i dag. For å kunne benytte seg av sterk kryptografi som Kerberos tilbyr må en applikasjon eller et system "kerberiseres" dvs. tilpasses bruk av Kerberos. Kerberos brukes bl.a. i Windows 2000/XP, Eudora, ftp, rlogin osv. Nær sagt alle applikasjoner som autentiserer brukere, kan bruke Kerberos dersom nødvendige tilpasninger blir gjort.

Kontinuerlig forbedring av protokoller De formelle metodene vi har sett på er ofte godt egnet til å kvalitetsikre utviklingen av en autentiseringsprotkoll under designprosessen. Feil som ikke oppdages under design og som forplanter seg til implementasjonen kan være veldig vanskelige å finne. Dessverre er det slik at en protokoll aldri kan regnes som 100% sikker, det sniker seg inn små spissfindige feil slik vi ser det i kap. 5. Protokoller som Kerberos blir stadig utviklet som svar på nye feil som oppdages og nye angrep som kan rettes mot de. Det er derfor nødvendig med stadig vedlikehold for å gi brukerne ønsket grad av sikkerhet. I arbeidet med kontinuerlig forbedringer av autentiseringsprotkoller kan AsmL være et nyttig verktøy, en retning som derfor er veldig interessant å studere. Hvordan kan vi bruke AsmL i arbeidet med vedlikehold og forbedring av autentiseringsprotokoller? Arbeidet med vedlikehold gjøres enklere ved bruk av kjørbare spesifikasjoner som AsmL, hvor en hele tiden kan se hvilke følger ny funksjonalitet får. Protokollen må hele tiden kunne utvides. Vi starter derfor med å innføre ny funksjonalitet og feilrettinger i AsmL–modellen og tester denne, før vi innfører det i den implementerte versjonen.

10.2.1 Testing av en AsmL spesifikasjon

En litt annen retning er å legge til rette for å bruke testmodulen som er utviklet for AsmL. Testdelen av AsmL består av en nyutviklet applikasjon (asmlt) som kan benyttes til å generere tester av AsmL–koden. Testmodulen kan brukes til modellbasert testing ved hjelp av semi-automatisk parametergenerering, generering av kall-sekvenser, generering av en endelig tilstandsmaskin og samsvarstesting. En slik automatisert testing vil lette arbeidet med utviklingen av nye applikasjoner og systemer. Dersom en har en modell som stadig utvikles, vil en slik automatisert testing være en svært verdifull hjelp når en kommer til testfasen i utviklingssyklusen. Per i dag er testgenereringen semi-automatisk, dvs. at den trenger hjelp fra bruker for å velge ut de mest interessante testene å kjøre. For et eksempel på slik testgenerering kan en se på testingen av en nettbutikk slik vi finner det hos (Barnett et al. 2003). Den endelige tilstandsmaskinen som asmlt genererer, kan "utforskes" på samme måte som modellsjekkere gjør. Det finnes ulike måter å beskjære søkerommet slik at testene er mulige å kjøre, selv om modellen gir et uendelig søkerom. Samsvarstesting kan utføres mot en ferdig implementasjon, for å teste om den nøyaktig følger modellen som er laget.

Kapittel 11

Konklusjon

I denne oppgaven har vi sett på autentiseringsprotokoller og formelle metoder for undersøkelsen av de. Vi startet i kapittel 2 med å se på noen essensielle sikkerhetstjenester og hvordan de er nødvendige for mange av de tjenestene som vi benytter oss av daglig. Videre så vi i kapittel 3 på hva kryptering er. Hvorfor er kryptering en forutsetning for sikkerhetstjenestene vi bruker, og mer spesielt hvordan kan de hjelpe oss i å autentisere hverandre over et datanettverk. Vi så hvordan kryptering har utviklet seg gjennom historien, fra kun å tilby meldingskonfidensialitet til å kunne gi oss autentisering over datanettverk, digitale signaturer, offentlige nøkkelkryptografisystemer m.m. I kapittel 4 så vi på protokoller, vi beskrev hva en protokoll er, og så på noen eksempler på ulike typer av protokoller. Vi kom også med en kort beskrivelse av hva en autentiseringsprotokoll er. Kapittel 5 ser på de angrep som kan rettes mot protokollgjennomføring, og hvilke feil som går igjen. Vi fikk en oversikt over hvilke typer av feil og angrep som opptrer. Det kan gi oss innsikt i hva det er vi prøver å unngå i arbeidet med analyse og utvikling av autentiseringsprotokoller. Vi så også kort på hva vi mener med en angriper og hvilke ressurser og egenskaper vi tillegger en slik angriper i et datanettverk.

I kapittel 6 bruker vi det vi har lært så langt om autentiseringsprotokoller og får en oversikt over de formelle metoder som er benyttet i arbeidet med autentiseringsprotokoller. Vi ser på de ulike typene av formelle metoder som finnes og beskriver fordeler og ulemper ved de. Vi gjør også en vurdering av de formelle metodenes bruksområde, hvilke metoder passer best til hva, og hvordan kan vi sikre oss best resultater. Noen metoder passer best i utviklingen av nye protokoller, andre egner seg bedre til analyse og verifikasjon. For en komplett behandling av en autentiseringsprotokoll er hybridsystemer å foretrekke, da vi på den måten kan behandle ulike egenskaper ved protokoller med den metoden som passer best. Vi har altså ikke *en* formell metode som er best, men flere som er bra på ulike nivåer. Det neste kapittelet er en beskrivelse av en velkjent autentiseringsprotokoll, Needham-Schroeder protokollen. Vi går detaljert igjennom den originale protokollen for å få et godt utgangspunkt for å lage en spesifikasjon av den, og for å se nærmere på kritikk som er rettet mot den. Kritikken blir beskrevet, og vi ser hva som er gjort for å rette opp i de feil som blir funnet. Tiden som går mellom beskrivelsen av den originale protokollen i

1978 frem til et nytt angrep som blir oppdaget i 1995 (17 år) viser hvor vanskelige det er å finne feil i slike protokoller. Det på tross av at protokollen er veldig enkel, sammmenlignet med dagens komplekse protokoller og programmer. Noe av grunnen er at vi må ta med en angriper i vurderingen, slik vi beskrev i kapittel 5.

Kapittel 8 beskriver den formelle metoden som vi benytter oss av i spesifikasjonen som presenteres i kapittel 10. Den formelle metoden vi benytter er Abstrakte tilstandsmaskiner (ASM). Vi ser nærmere på metoden og det teoretiske fundamentet den bygger på. Vi beskriver hvordan ASM kan brukes i utviklingen av nye protokoller, programmer og maskinvare. Videre ser vi i kapittel 9 et kjørbart spesifikasjonsspråk kalt Abstrakt tilstandsmaskinspråk som vi bruker til å kode spesifikasjonen i kapittel 10. Vi ser kort på AsmL som språk, hvordan det blir brukt og hvilke muligheter det gir. Spesifikasjonen i kapittel 10 blir så gjennomgått, og vi avslutter med å se på hvilke retninger videre arbeid kan føres i. Vi kan se på en utvidet versjon av protokollen, vi kan behandle kryptering, kommunikasjon og testing i større detalj. ASM og AsmL viser seg å være godt egnede metoder for utvikling og vedlikehold av spesifikasjoner. Meadows sier i sin artikkel "Open issues in formal methods for cryptographic protocol analysis" at det er behov for verktøy og metoder som kan undersøke "skisser" av autentiseringsprotokoller nøye, før de blir presentert som et ferdig produkt (Meadows 2001). Vi har sett hvordan dette kan gjøres med spesifikasjonen av Needham-Schroeder, og ser at det er gode muligheter for slik bruk av ASM og AsmL. Ved hjelp av de verktøy som AsmL gir oss, kan vi avdekke feil og mangler før vi implementerer. AsmL verktøyene støtter også testing i utviklingsprosessen og kan være svært nyttige i vedlikehold av et ferdig produkt.

Tillegg A

AsmL koden

Her kommer AsmL koden med kommentarer slik den er definert i fila *Model1.asml*.

A.1 Innledning

Først er det nødvendig å finne ut av hva som trengs for å lage en god formell modell av Needham-Schroeder (N-S) protokollen for autentisering. Vi må også gjøre noen valg i forhold til hvilken utgave/versjon av den vi ønsker se nærmere på. I sin kjente artikkel: "Using Encryption for Authentication in Large Networks of Computersbeskriver Roger M. Needham og Michael D. Schroeder en tenkt protokoll for autentisering i store datanettverk. Det finnes to versjoner av protokollen avhengig av om det er involvert en pålitelig tredjepart (server) eller ikke.

A.2 Needham-Schroeder protokollen

Vi har flere ulike inkarnasjoner av N-S protokollen. Først ser vi på den som involverer en tredjepart og benytter seg av offentlig nøkkelkryptering. Da ser den slik ut:

1. A -> S : A,B
2. S -> A : Kb, BK s -1
3. A -> B : Na, AK b
4. B -> S : B, A
5. S -> B : Ka, AK -1
6. B -> A : Na, NbK a
7. A -> B : NbK b

En kort forklaring av hva som skjer mellom linjene:

- I (1.) sender A en forespørsel til en (autentiserings/nøkkel) server S om å kommunisere med B ved å inkludere sin egen og Bs identitet.
- (2.) A får en melding tilbake fra S som består av en sesjonsnøkkel for kommunikasjon med B og B sin identitet alt kryptert med S sin private nøkkel, noe som gir en signatur og integritetssjekk av meldingen fra S.
- (3.) A sender så sin identitet sammen med en engangsverdi (nonce) generert av A til B, kryptert med B sin offentlige nøkkel som A mottok fra S.
- (4.) B sender så sin egen og As identitet til S. Dette for å si til S at B trenger A sin offentlige nøkkel.
- (5.) S sender tilbake til B, A sin offentlige nøkkel, As identitet kryptert med S sin private nøkkel for å forsikre B om opprinnelsen til meldingen.
- (6.) Nå kan B sende Na, Nb til A, kryptert med A sin offentlige nøkkel. Dette er et svar på forespørselen om å opprette autentisert kommunikasjon mellom A og B.
- (7.) I den siste meldingen som gjør autentiseringen komplett, sender A, Nb tilbake til B som svar på melding 6.

Den andre versjonen ligner veldig på den forrige, men den benytter seg av symmetrisk krypto og nøkler som er utvekslet tidligere på en sikker måte, og den fungerer som følger:

1. A -> S : A, B, Na
2. S -> A : Na, B, K, K,AE b E a
3. A -> B : K, AE b
4. B -> A : NbE k
5. A -> B : Nb-1E k

Vi har også en veldig enkel versjon av protokollen, som jeg tar utgangspunkt i når jeg nå skal lage en formell modell:

1. A -> B : Na, AKb
2. B -> A : Na, Nb, BKa
3. A -> B : NbKb

Den forutsetter at agentene på forhånd har utvekslet sine offentlige kryptonøkler. Protokollen er egentlig bare melding 3, 4 og 7 fra den første protokollen beskrevet over. Se også G. Lowes artikkel: "An attack on the Needham-Schroeder public key authentication protocol.De er også endel ting som skjer mellom linjene som må behandles. Mellom linje 1 og linje 2 så skal agent B motta og forstå innholdet i meldingen fra A. Det vil si at agent B må dekryptere meldingen, og utifra innhold og rekkefølge forstå at dette er agent A som ønsker å autentisere seg overfor agent B. Agent B må også sjekke at meldingen er kryptert med B sin offentlige nøkkel, og gjør dette ved å dekryptere meldingen med sin egen private nøkkel.

A.3 Objekter og klasser

Vi starter med å deklarere et namespace NeedhamSchroeder.

```
namespace NeedhamSchroeder
```

Hvilke datastrukturer er nødvendige for å formalisere Needham-Schroeder protokollen? Dersom vi ser på den siste (og enkleste) versjonen ser vi at det er nødvendig med klasser og objekter som kan representere agenter, nøkler, nonces og meldinger.

- Vi har flere agenter, med navn, som skal kommunisere med hverandre. De har tilgang til en liste med hvilke (offentlige) nøkler de har tilgjengelig.
- Vi trenger en beskjed som består av: identitet(er) og nonce. Vanligvis er dette kryptert med en offentlig nøkkel slik som i vårt tilfelle.
- For å kunne realisere dette trenger jeg klasser for de ulike objektene som opprettes i eksekveringen av modellen. Vi trenger i første omgang Agent, Nonce, Nøkkel og Beskjed.
- Det er også nødvendig med noen funksjoner: en krypteringsfunksjon og en funksjon som gir meg tilfeldige tall. Disse tilfeldige tallene brukes som nonce.

A.4 Agent

En agent representerer en bruker eller et program som skal operere i et datanettverk. Agenten skal kunne opprette sikre kommunikasjonskanaler med andre agenter ved hjelp av en autentiseringsprotkoll. Etter at autentisering mellom partene er foretatt, er det mulig å utveksle nøkler for kryptert kommunikasjon. Agenten består i modellen av en klasse med følgende felter: navn (identitet), nøkler (de offentlige nøklene agenten har), tilstand (for å angi hvilken tilstand agenten befinner deg i til enhver tid), nonces (hvilke nonces agenten har mottat) og en mailbox hvor beskjeder fra andre agenter kommer inn. Som vi kan se er det forskjellige variablene deklarert vha. til ulike innebygde typer i AsmL. Vi kommer

tilbake til de forskjellige typene, men de er hovedsakelig slik vi kjenner de også fra andre programmeringsspråk.

```
public class Agent
  var navn as String
  var nøkler as Map of String to Integer
  var tilstand as String
  var mailbox as Seq of Beskjed = []
  var nonces as Seq of Nonce = []
```

A.5 Nonce

Nonce er en engangsverdi som kun skal brukes engang. Nonce blir generert av en av agentene i protokollen. Nonce er realisert som en klasse med en verdi (et randomgenerert tall) og en identitetsvaiabel som angir hvilken agent som genererte verdien.

```
sealed public structure Nonce //Structuren kan ikke forandres på
    siden
                              //den er "sealed", har med tilgang
                                  til metoden å gjøre.
  verdi as Integer
  ident as String
```

A.6 Nøkler

Nøklene som brukes er av typen offentlige nøkler (ON) eller private nøkler (PN). Vi har en overordnet klasse Nøkkel. Videre i arbeidet med protokollen forutsetter vi at vi har egne funksjoner som gjør arbeidet med kryptering og dekryptering på en sikker og god måte. Det finnes mange eksempler på gode krypteringsalgoritmer. Vi tar derfor ikke med en implementasjon av disse i oppgaven. Derfor er det kun nødvendig å sjekke om en agent har tilgang til riktig nøkkel i modellen. Dette gjøres vha. funksjonen Kryptering?. Jeg tar det også for gitt at riktig nøkkel er benyttet ved kryptering.

```
public class Nøkkel
```

A.7 Tidsstempel

Tidsstempel, inneholder en eller annen form for tidsangivelse. Er ikke nødvendig i vår modell, men kan brukes i eventuelle utvidelser av modellen, se også Denning og Saccos angrep på N-S protokollen og deres løsnig på dette problemet ved hjelp av Tidsstempler. Denning og Sacco: Timestamps in Key Distribution Protocols".

Tidsstempelet må inneholde tid i et eller annet format, kanskje en integer som gir antall sekunder. (her må det tas hensyn til hvilken presisjon som er nødvendig for tidsangivelsen) Her er det også mulig å benytte seg av funksjoner importert fra C# kanskje?

```
public class Tidsstempel
    var tid as Integer
```

A.8 Kryptering og dekryptering

Vi trenger også funksjoner som tar for seg selve kryptering/dekrypteringen. Deklarer også noen egne typer slik at jeg kan bruke de i funksjonene. Kryptering sjekker om en gitt beskjed lar seg dekryptere, dvs. om agenten har tilgang på riktig nøkkel for dekryptering.

```
type beskjed = Beskjed
type nøkkel = Nøkkel
type nonce = Nonce
type agent = Agent

function Kryptering(i as beskjed, j as String) as Boolean
  if j in i.nøkkel then
    true
  else
    false
```

A.9 Beskjeder

Id er identiteten til en agent gitt som en String. Feks. "Aliceeller Bob". Beskjed består av de feltene som er nødvendige for å kunne sende de ulike beskjedene vi finner i modellen. Vi har nonce1, nonce2, id og nøkkel. Hvor nonce1 og 2 er respektive nonces, i noen beskjeder vil det være bare en nonce. Id er identiteten til en av agentene. Nøkkel er den nøkkelen som er brukt for å kryptere beskjeden.

```
 sealed public class Id
   var i as String


public class Beskjed
    var nonce1 as Nonce
    var nonce2 as Nonce or Null
    var id as String
    var nøkkel as Set of String
```

A.10 Metoder og funksjoner til beuk i kjøring av protokollen

Vi har også noen ekstra funksjoner for å hjelpe oss i kjøringen av modellen. OppdaterTilstand er en funksjon som avhengig av tilstanden som en agent befinner seg i oppdaterer tilstanden til agenten. Jeg har kun tatt med akkurat de tilstandene en agent må gå igjennom for å bli autentisert. I selve kjøringen sjekkes tilstand, og en agent vil ikke gå til neste tilstand uten at alt er slik det skal være i forrige tilstand. LeggTilNonce vil legge til en nonce som er mottat i en beskjed i nonces hos agenten. Dette gjøres etter at det er utført en sjekk på om meldingen er kryptert riktig. SjekkNonce skal brukes til å sjekke om den nonce som A sendte til B i beskjed 1, er den samme som den nonce A får tilbake fra B i beskjed 2.

```
function OppdaterTilstand (a as Agent, t as String)
  if t = "start" then
    a.tilstand := "1"
  elseif
    a.tilstand = "1" then
    a.tilstand := "2"
  elseif
    a.tilstand = "2" then
    a.tilstand := "Autentisert"


function LeggTilNonce (a as Agent, n as Nonce)
  a.nonces := [n]

function SjekkNonce (a as Nonce, b as Nonce)as Boolean
  if b = a then
    true
  else
    false
```

A.11 Rammeverk for kommunikasjon mellom agenter

Vi trenger et rammeverk som hjelper oss med å få agentene som blir opprettet til å kommunisere. Kommunikasjonen foregår ved utveksling av beskjeder. Disse beskjedene er de samme som de vi har definert i klassen Beskjed. Jeg prøver her å bruke noe av koden som er skissert i Glässer et al. sin artikkel: "An abstract communication model". De tenker seg at hver agent har en mailbox som de andre agentene kan putte meldinger i. Jeg legger til den nødvendige mailboxen for dette i Agent klassen. I tillegg benytter jeg meg av funksjonen SendMessage, den putter beskjeden fra en agent inn i mailboxen til mottageragenten som spesifiseres i funksjonskallet.

```
  SendMessage(a as Agent, m as Beskjed)
2   a.mailbox := [m]
```

A.12 Main() - hoveddel med eksekvering

Her kommer selve eksekveringen av modellen. Her benyttes de ulike klasser og funksjoner som er deklarert ovenfor for å realisere en eksekvering av modellen. Modellen starter med at to agenter Alice og Bob initialiseres og videre ser vi hvordan de kan gjennomføre en autentisering ved hjelp av protokollen som modellen realiserer. Se neste side.

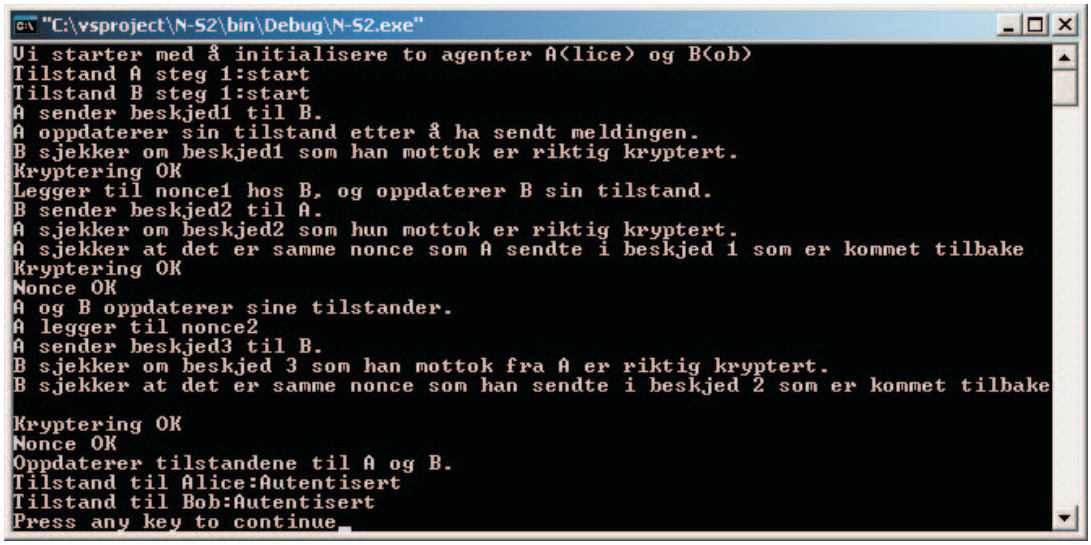
```
2  [EntryPoint]
   Main()
4    var a = new Agent("Alice",{"On(B)" -> 1},"start")              //Initialiserer to agenter
       Alice og Bob.
     var b = new Agent("Bob",{"On(A)" -> 2},"start")
6    var beskjed1 = new Beskjed(Nonce(GetRandomSeed(), "A"),null,"A",{"OnB"})

8    step
       WriteLine("Vi starter med å initialisere to agenter A(lice) og B(ob)")
10     WriteLine("Tilstand A steg 1:" + a.tilstand)
       WriteLine("Tilstand B steg 1:" + b.tilstand)
12     WriteLine("A sender beskjed1 til B.")
       SendMessage(b, beskjed1)                                    //Sender beskjed1 til Bob fra Alice
14
       OppdaterTilstand(a, "start")                                //Oppdaterer Alice sin tilstand fra start
         til 1
16     WriteLine("A oppdaterer sin tilstand etter å ha sendt meldingen.")
     step
18     let mb = Head(b.mailbox)                                    //plukker ut meldingen fra Bob sin mailbox
         til en lokal variabel for enkel tilgang
       WriteLine("B sjekker om beskjed1 som han mottok er riktig kryptert.")
20
       if Kryptering(mb, "OnB") then                               //sjekker om kryptering er gjort med riktig
         nøkkel
22       WriteLine("Kryptering OK") else
         error "Feil nøkkel"
24
```

```
    step

26

      LeggTilNonce(b, mb.nonce1)                    //legger til nonce fra Alice hos Bob.
28    OppdaterTilstand(b, "start")                  //Oppdaterer tilstanden til Bob.
      WriteLine("Legger til nonce1 hos B, og oppdaterer B sin tilstand.")
30    step

32    var beskjed2 = new Beskjed(Head(b.nonces),Nonce(GetRandomSeed(), "B"),"B",{"OnA"})     //
          beskjed 2 som Bob sender til Alice


34

    step

36

      SendMessage(a, beskjed2)
38  /*Bob sender beskjed 2 med Alice sin nonce og Bob sin nye nonce til Alice */
      WriteLine("B sender beskjed2 til A.")
40    WriteLine("A sjekker om beskjed2 som hun mottok er riktig kryptert.")
      WriteLine("A sjekker at det er samme nonce som A sendte i beskjed 1 som er kommet tilbake")
42    step
      let mba = Head(a.mailbox)
44    let noncea = mb.nonce1
      let nonceb = mba.nonce1

46

      //WriteLine("Nonce:" + noncea) test av let binding av nonce

48

      if Kryptering(mba, "OnA") then                //sjekker om melding kryptert med Alice sin
          off.nøkkel
50      WriteLine("Kryptering OK") else
```

```
      error "Feil nøkkel"
52
    if SjekkNonce(noncea, nonceb) then          //sjekker om melding inneholder nonce som A
      WriteLine("Nonce OK") else                 // sendte til B i beskjed1.
      error "Feil nonce"
56

    OppdaterTilstand(b, "1")                    //Oppdaterer tilstanden til Bob
    OppdaterTilstand(a, "1")                    //Oppdaterer tilstanden til Alice
60
    WriteLine("A og B oppdaterer sine tilstander.")
62  step
    LeggTilNonce(a, mba.nonce2)                 //legger til noncen fra Bob hos Alice
64  WriteLine("A legger til nonce2")

66  step
    var beskjed3 = new Beskjed(Head(a.nonces),null,"A", {"OnB"})
68  SendMessage(b, beskjed3)
    WriteLine("A sender beskjed3 til B.")
70  step
    let mbb = Head(b.mailbox)
72  let noncea1 = mba.nonce2
    let nonceb1 = mbb.nonce1
74


76


78
```

```
    WriteLine("B sjekker om beskjed 3 som han mottok fra A er riktig kryptert.")
80  WriteLine("B sjekker at det er samme nonce som han sendte i beskjed 2 som er kommet tilbake
    ")

82  if Kryptering(mbb, "OnB") then                    //sjekker om melding kryptert med Bob sin off
    .nøkkel
      WriteLine("Kryptering OK") else
84    error "Feil nøkkel"

86  if SjekkNonce(noncea1, nonceb1) then               //Sjekker om melding3 inneholder
      WriteLine("Nonce OK") else                      // nonce som B sendte til A i beskjed3
88    error "Feil nonce"

90  step
      OppdaterTilstand(a,"2")
92    OppdaterTilstand(b,"2")
      WriteLine("Oppdaterer tilstandene til A og B.")
94  step
      WriteLine("Tilstand til Alice:" + a.tilstand)
96    WriteLine("Tilstand til Bob:" + b.tilstand)
```

A.13 Kjøringseksempel



Figur A.13.1: Kjøringseksempel for N-S i AsmL.

A.14 AsmL kode i nettleser

Figur A.14.1: AsmL kode i nettleser

Figurer

Referanser

Abstract state machine language.

Abadi, M. og R. Needham (1996). Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering 22*(1), 6–15.

Anderson, R. og R. Needham (1995). Programming satan's computer.

Anlauff, M. (1998, November). Abstract state machines an introductory tutorial. ppt-slide as html. SCCC98, Antofagasta, Chile.

Barnett, M., W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, og M. Veanes (2003, October). Towards a tool environment for model-based testing with asml. web. 3rd International Workshop on Formal Approaches to Testing of Software (FATES 2003).

Barnett, M. og W. Schulte (2003, March). Runtime verification of .net contracts. *Journal of Systems and Software 65*, 199–208.

Bella, G. (2001). Lack of explicitness strikes back. *Lecture Notes in Computer Science 2133*, 87–??

Bella, G. og E. Riccobene (1997). Formal analysis of the Kerberos authentication system. *J.UCS: Journal of Universal Computer Science 3*(12), 1337–??

Bella, G. og E. Riccobene (1998). A Realistic Environment for Crypto-Protocol Analyses by ASMs. I *Proceedings of the 28th Annual Conference of the German Society of Computer Science*. Technical Report, Magdeburg University.

Bishop, M. (2003). *Computer Security - Art and Science*. Addison-Wesley.

Börger, E. og R. Stärk (2003). *Abstract State Machines A method for High-Level System Design and Analysis*. Springer-Verlag.

Burrows, M., M. Abadi, og R. Needham (1990, February). A logic of authentication. *ACM Transactions on Computer Systems 8*(1), 18–36.

Buttyan, L. (1999). Formal methods in the design of cryptographic protocols (state of the art).

Carlsen, U. (1994). Cryptographic protocol flaws, know your enemy. I *Proceedings 7th IEEE Computer Security Foundations Workshop*, s. 192–200. IEEE.

DeMillo, R. A., N. A. Lynch, og M. J. Merritt (1982). Cryptographic protocols. I *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, s. 383–400. ACM Press.

Denker, G., J. Meseguer, og C. Talcott (1998). Protocol specification and analysis in Maude. I *In Proc. of Workshop on Formal Methods and Security Protocols*.

Denning, D. E. og G. M. Sacco (1981, August). Timestamps in key distribution protocols. *Communications of the ACM 24*(8), 533–536.

Diesen, D. (1995, Mars). *Specifying Algorithms Using Evolving Algebra. Implementation of Functional Programming Languages*. D. sc. thesis, Universitetet i Oslo, Norge.

Dolev, D. og A. Yao (1983, March). On the security of public key protocols. *IEEE Transactions on Information Theory 29*(2), 198 – 208.

EFF (1998). *Cracking DES,Secrets of Encryption Research, Wiretap Politics & Chip Design*. EFF and O'Reilly and Associates.

Fagin, R., J. Y. Halpern, Y. Moses, og M. Y. Vardi (1995). *Reasoning about knowledge*. MIT Press.

Foundations of Software Engineering Microsoft Research. *AsmL: The Abstract State Machine Language Reference*. Foundations of Software Engineering Microsoft Research.

Foundations of Software Engineering Microsoft Research. *Introducing AsmL: A Tutorial for the Abstract State Machine Language*. Foundations of Software Engineering Microsoft Research.

Glässer, U., Y. Gurevich, og M. Veanes (2002, May). An abstract communication model. Technical Report MSR-TR-2002-55, Microsoft Research, One Microsoft Way Redmond, WA 98052.

Gong, L. (1990). Verifiable-text attacks in cryptographic protocols. I *Proceedings of IEEE INFOCOM '90*, s. 686–693. IEEE: IEEE Computer Society Press.

Gritzalis, S. og D. Spinellis (1997). Cryptographic protocols over open distributed systems: A taxonomy of flaws and related protocol analysis tools. I *16th International Conference on Computer Safety, Reliability and Security: SAFECOMP '97*, Berlin, Germany / Heidelberg, Germany / London, UK / etc., s. 123–137. Springer Verlag.

Gurevich, Y. (1993). Evolving algebras: An attempt to discover semantics. I G. Rozenberg og A. Salomaa (utg.), *Current Trends in Theoretical Computer Science*, s. 266–292. River Edge, NJ: World Scientific.

Gurevich, Y. (1994). Evolving algebras 1993: Lipari Guide. I E. Börger (utg.), *Specification and Validation Methods*, s. 9–37. Oxford University Press.

Huggins, J. K. og C. Wallace (2002, December). Asm 101: An abstract state machine primer. Technical Report CS-TR-02-04, Departement of Computer Science, Michigan Technological University, Houghton, MI 49931-1295.

Jervell, H. R. (2001). *Logikk og beregnbarhet*. Unipub forlag.

Johnsen, B. (2001). *Kryptografi - den hemmelige skriften - Kryptografiens kulturhistorie fra år 0 til 2001*. Tapir Akademsik forlag.

Knuth, D. E. (1984, May). Literate programming. *The Computer Journal 27*(2), 97–111.

Lowe, G. (1995, August). An attack on the needham-schroeder public-key authentication protocol.

Lowe, G. (1996). Breaking and fixing the Needham-Schroeder public-key protocol using FDR. I *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Volume 1055, s. 147–166. Springer-Verlag, Berlin Germany.

Lowe, G. (1997). A family of attacks upon authentication protocols.

Massey, J. L. (1988, May). An introduction to contemporary cryptology. *Proceedings of the IEEE 76*(5).

Meadows, C. (2001). Open issues in formal methods for cryptographic protocol analysis. *Lecture Notes in Computer Science 2052*, 21+.

Meadows, C. A. (1994). Formal verification of cryptographic protocols: A survey. I *ASIACRYPT: Advances in Cryptology – ASIACRYPT: International Conference on the Theory and Application of Cryptology*. LNCS, Springer-Verlag.

Needham, R. M. og M. D. Schroeder (1978a, December). Using encryption for authentication in large networks of computers. *Communications of the ACM 21*(12), 993–999.

Needham, R. M. og M. D. Schroeder (1978b, December). Using encryption for authentication in large networks of computers. *Communications of the ACM 21*(12), 993–999.

Robert F. Stärk, J. S. og E. Börger (2001). *Java and the Java Virtual Machine Definition, Verification, Validation*. Springer-Verlag.

Rosenzweig, D., D. Runje, og N. Slani (2003, March). Privacy, abstract encryption and protocols: an asm model - part i. I E. Börger, A. Gargantini, og E. Riccobene (utg.), *Abstract State Machines - Advances in Theory and Applications: 10th International Workshop*, Volume 2589 of *Lecture Notes in Computer Science*, Taormina, Italy, s. 372–390. Springer-Verlag.

Rubin, A. D. og P. Honeyman (1993). Formal methods for the analysis of authentication protocols. Technical Report CITI Technical Report 93-7, CITI University of Michigan.

Ryan, P. og S. Schneider (2001). *The Modelling and Analysis of Security Protocols*. Addison-Wesley.

Schneier, B. (1996). *Applied Cryptography Second Edition: protocols, algorithms, and source code in C*. John Wiley & Sons, Inc.

Schneier, B. (2000). *Secrets & Lies, Digital Security in a Networked World*. John Wiley & Sons, Inc.

Schneier, B. (2002, January). Windows upnp vulnerability. html. CryptoGram Newsletter.

Singh, S. (1999). *The Code Book - The secret history of codes and code-breaking*. Fourth Estate.

Smith, R. E. (2002). *Authentication From Passwords to Public Keys*. Addison-Wesley.

Snekkenes, E. (1992, May). Roles in cryptographic protocols. I *Proceedings of the IEEE Computer Society Symposium on Security and Privacy*, s. 108–113.

Stallings, W. (2003). *Network Security Essentials, Applications and Standards*. Prentice Hall, Pearson Education International.

Syverson, P. og C. Meadows (1993). A logical language for specifying cryptographic protocol requirements. I *RSP: IEEE Computer Society Symposium on Research in Security and Privacy*.

Syverson, P., C. Meadows, og I. Cervesato (2000). Dolev-yao is no better than machiavelli.

Uwe Glässer, Y. G. og M. Veanes (2002, May). An abstract communication model. Technical Report MSR-TR-2002-55, Microsoft Research, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052.

Whitfield Diffie, Paul C. van Oorschot, M. J. W. (1992, June). Authentication and authenticated key exchanges. *Designs, Codes and Cryptography 2*(2), 107–125.

Xu, C., G. Kedem, og F. Gong (2000). Categorizing attacks on cryptographic protocols based on intruders' objectives and roles.

A.15 Kommentar til bibliografien

Jeg har i løpet av arbeidet med oppgaven lest mange artikler og bøker. Jeg vil kort nevne de som jeg har brukt mest, da referanselisten er lang. I forbindelse med kapittel 2 har jeg hatt stor nytte av bøkene *Secrets and Lies* av Schneier (2000)og *Computer Security* av Bishop (2003). Generell informasjon om sikkerhet og kryptering i distribuerte nettverk har hovedsaklig kommet fra *Network Security Essentials* av Stallings (2003) og klassikeren *Applied Cryptography* av Schneier (1996). Jeg har også lest en norsk bok med et historisk perspektiv; *Kryptografi - den hemmelige skriften - Kryptografiens kulturhistorie fra år 0 til 2001* av Johnsen (2001). I kapittelet om angrep og feil har jeg benyttet noen artikler som gir en oversikt bl.a. fra Carlsen (1994) og Meadows (1994). Angriperen beskrives hos Dolev og Yao (1983). De formelle metodene finnes i et utall av artikler, Buttyan (1999) gir en oversikt og artikkelen om BAN–logikk er en klassiker Burrows, Abadi, og Needham (1990). Jeg har hatt stor nytte av artikkelen til Lowe om det nyoppdagede angrepet på Needham-Schroeder protokollen (Lowe 1995). Artikkelen til Needham og Schroeder har også vært

viktig i arbeidet med protokollen (Needham og Schroeder 1978b). ASM kapittelet henter mesteparten av innholdet sitt fra *The ASM book* Börger og Stärk (2003) og fra Gurevich (1994). En bok som har gitt meg et overblikk over ulike typer av autentisering har vært *Authentication From Passwords to Public Keys* av Smith (2002).